Zissos

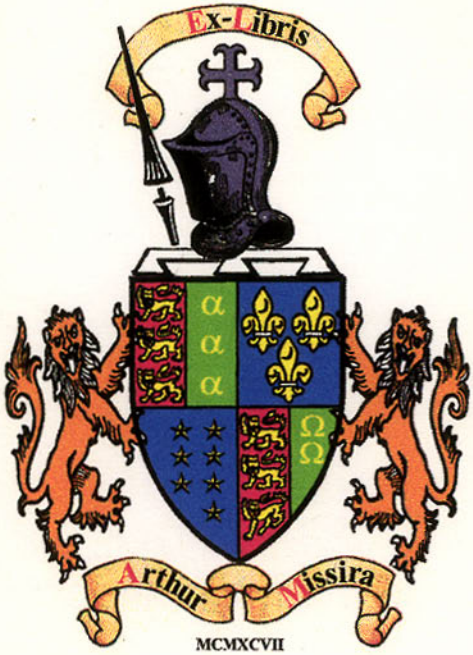
System Design with Microproces

System Design with Microprocessors

D. Zissos

Academic Press London New York San Francisco
A Subsidiary of Harcourt Brace Jovanovich, Publishers

System Design with Microprocessors



SYSTEM DESIGN WITH MICROPROCESSORS

D. ZISSOS

Department of Computer Science, University of Calgary,
Calgary, Canada

with contributions by

J. C. BATHORY,

Research Assistant, University of Calgary

1978

ACADEMIC PRESS

London New York San Francisco

A Subsidiary of Harcourt Brace Jovanovich, Publishers

ACADEMIC PRESS INC. (LONDON) LTD.
24/28 Oval Road
London NW1

United States edition published by
ACADEMIC PRESS INC.
111 Fifth Avenue
New York, New York 10003

Second Printing 1979

Library of Congress Catalog Card Number: 78-54544
ISBN: 0-12-781750-6

Printed in Great Britain by
Willmer Brothers Limited, Birkenhead

Foreword

THE IMPACT OF THE NEW TECHNOLOGY ON COMPUTING

There is no doubt that the recent developments in technology, which are discussed in this book, will have a profound impact on computing, but as yet there is no consensus of opinion on what that impact might be. The Computer Board for Universities and Research Councils which authorises and monitors all major items of computing equipment in Universities has a significant interest in the effect of the new technology on the computing environment. It therefore commissioned a Working Party under my chairmanship to examine the issues involved and to assess the likely affect on computing style, provision and facilities. The task of this Working Party is an on-going one but I shall summarize the opinions which I have formulated from the extensive discussions which have already been carried out.

It is important to maintain a perspective on the main issues involved. Far too often statements are made in the computing press which foretell the demise of computer centres and the impending obsolescence of networks. Proponents of such views predict a rapid shift to dispersed computing with each department, section and even individual members of an organization having their own powerful computing system on the desk for a few pounds. It is my intention to show that whilst there will be a shift towards dispersed computing, far more fundamental consequences will result from expansion into new computing areas and the introduction of computers to wide sections of the community.

The first and perhaps most important point is a consequence of the equal importance of both hardware and software in the computing environment, and this book properly emphasizes both. The dramatic fall in the costs of power, main storage, and some secondary storage devices, has been paralleled with escalating software and service costs. Even today the recurrent costs arising from software provision and user services far outweigh any hardware

costs. Indeed when all costs are averaged over the life of a medium or large scale computer installation they are found to comprise about 80% software, service and maintenance costs, and only 20% or thereabouts in the hardware alone. If we take the limit and assume that hardware will eventually be free, total costs will only reduced by about 20%. Thus the new technology is not going to make large or medium scale computing any cheaper for the traditional computer centre. But how will the costs of dispersed computing be affected?

It is important to realise that the new technology will permeate society in two ways—the high volume/low cost and low volume/high cost categories. The former type will be produced in millions and will be very cheap indeed. They will often be single chips dedicated to one application and, as such, will not even be recognizable as computers. Such chips will be found in everyday equipment such as cars, washing machines and TV sets. They will have no peripheral equipment (apart from signal converters) and will probably not even be maintained in the conventional computer sense. High sales volumes will be required to keep production costs down. Indeed expansion into new areas will be essential to the success of the new microprocessor industry because the existing data processing market will not provide a viable market place for such chips. It has been estimated for example, that the whole of the data processing capability of the world in 1978 represented two weeks production capacity for the chip manufacturers. Of course the data processing market will benefit incidentally from the creation of the new markets but it will not be the driving force. Design in this new environment will be market driven rather than architecture driven.

The data processing market place will comprise the low volume/high cost category mentioned earlier. Higher level units will be constructed from the chips with a corresponding increase in costs. Such systems, in terms of hardware will still be cheap, but the high cost of mechanical peripherals will keep prices relatively high. The new technology will however enable special purpose devices such as Distributed Array Processors, or Data Base Engines to be constructed with superior performance compared with existing devices. In the future a whole range of systems will be available from small personal computers costing a few hundred pounds, to large systems with a power in excess of 50 Mips for, say, £2 million. Thus from a hardware standpoint the new technology will widen the range of facilities available at reduced costs.

Computer systems do not however consist of hardware alone. The software required to operate on the new technology will also be provided in the same two categories—high volume/low cost and low volume/high cost, and each category will tend to run on the equivalent category of hardware. The software in the single purpose chips for example will also be single purpose, relatively fault free, and very cheap if only because of the large volume of sales. It will be

unalterable by the user, and will not be maintained. It will simply have to work. The low volume/high cost software will be that mainly supplied to the data processing market place. The relatively low volumes involved will inevitably mean high prices. Because of such prices the user will demand maintenance which will raise prices even further. Thus the end user will see falling hardware prices but at best stable and probably rising software prices. Whilst the overall effect might be one of falling costs the era of really cheap computing predicted by so many, will simply not arrive.

There are however some potential dangers in the exploitation of the new logic in the data processing environment. Some users will be tempted to buy really cheap hardware and write their own software. At the present time such an approach is particularly hazardous because of the rapid changes taking place. The software available on the present generation of microprocessors is rather primitive, mainly at assembler or machine code level. Any potential do-it-yourself software programmer will require a host of additional hardware and software aids to enable him to properly debug his programs such as simulators, emulators, logic state analysers and development systems. It is not uncommon for such a user to outlay 5 or 10 times the cost of the initial system in further back-up equipment. Even then the software development is expensive because it is a human activity. Furthermore the advent of the 16-bit microprocessor to be followed within 2 years by the 32-bit system will render obsolete any back-up equipment purchased. There will, of course, be areas where user development is inevitable, but no one should underestimate the amount of effort involved, and I hope that those who read this book, and carry out software development will take careful note of the heavy demands upon their time.

There are other aspects of the computing environment which will tend to keep costs relatively high, even in a dispersed environment. One such example is file maintenance. Files are the basic material of any data processing environment. At present the rapid advances in disk design have rendered the dispersal of filestore as uneconomic. The billion byte disk is now about 2 years away and no collection of floppy disks or bubble stores will be able to match such storage techniques for many years. But file maintenance involves more than the costs of secondary storage. A computing centre with a large central filestore will be able to maintain a large collection of files much more economically than a host of end users in a dispersed system, and the security of files in a centralised approach (one or a few centres) must be higher than that of a fragmented dispersed system. The only environment in which a totally dispersed file store is meaningful would be that where user department files are unique and non-overlapping. But even where the overlap was minimal, careful thought would have to be given to transmission costs of transferring essential data between dispersed systems.

It is my view that the new computing environment will consist of a hierarchy of computing systems connected together via a network. Exactly how much power and filestore resides in a particular place will depend more on the Post Office networking charges than microprocessor costs. Any shift in tariffs will result in a shift in dispersion. In this view of the future, computing centres, networks, and personal computers all have their place. The main effects of the new technology will however be the extension of computing into new areas, and a widening of the community of people for whom the applications of computers are important. The extension of the user community, many of whom will have no expertise in computing, will require a significant change in our approach to software writing. The new software will have to be more tolerant of human frailty and much more user-orientated. I have called such software, sympathetic software. By a happy coincidence the technology which generates the requirement for such software also provides an economical basis for producing it. So often in the past the software designer has been severely constrained by having to be economical in his use of main storage. Now, not only can he use extra storage at little additional cost, he can also spread interpreting software round the hierarchy interpreting the same application requirements in different ways for different classes of user. Additionally, because interactive computing costs will reduce, he can also use the interactive dialogue with the inexperienced user to clarify any areas of doubtful interpretation, and also such software will need a 'help' facility to assist the user when he is perplexed.

I hope that in this foreword I have drawn your attention to some of the wider issues which result from the applications of the technology which is described and explained in Professor Zissos's excellent book. Let us now concern ourselves with presenting a better computer image to the new users by considering carefully the essential design characteristics of sympathetic software. If we fail in this objective, the anticipated rapid expansion of computing into new areas may not take place.

Liverpool J. L. ALTY
August 1978

Preface

The potential of the microprocessor is now universally recognized. However what is not so widely appreciated is that the device, which combines simplicity with versatility, can be used by those who have no knowledge of electronics but who wish to design and implement their own systems. Such users must, however, have access to formal step-by-step procedures and this book has been written in order that all those who wish to do so can design and implement systems using microprocessors. It aims to save microprocessors from the fate of computer technology in the 1950s and 1960s when great emphasis was placed on software design and minimum attention was paid to the formalization of hardware design. The result was a technology which, while affecting the lives of millions, was accessible to very few. Computers became shrouded in mystery and that shroud remains.

The opening chapter of this book introduces the reader to basic logic design—an essential prerequisite to the design and implementation of microprocessor systems.

Chapter 2 describes the microprocessor chip and emphasizes those features common to all microprocessor systems, as well as outlining the design philosophy used throughout the book. Subsequent chapters describe the various microprocessors systems; problems and their solution using these systems are included.

The average design cycle of a small microprocessor system is now about 50 minutes. It is not necessary to allow time for debugging, since, systems implemented using the steps described in this book, *always work*.

Those wishing to acquire more detailed information about microprocessor software are referred to a forthcoming book by F. G. Duncan entitled 'Microprocessor Programming and Software Design'. It will be published by Prentice-Hall in 1979.

The research upon which this book is based has been supported by a grant from the National Research Council of Canada.

Calgary
August 1978

D.Z.

Contents

1
Logic Design

In this chapter the basic concepts necessary to help the reader acquire a working knowledge of logic design are outlined. This knowledge is essential for the design and implementation of the hardware component of systems.

The design steps outlined are based on the use of *sequential equations* developed in 1969 by the author. All circuits implemented using these equations are hazard-free when constructed with gates of $\pm 33\frac{1}{3}\%$ maximum speed variation, and always work.

1.1 INTRODUCTION

Logic design is defined as a set of clear-cut step-by-step procedures used to implement realistically logic circuits given their I/O (input/output) characteristics. Logic circuits are classified as *combinational* and *sequential.* A *combinational* circuit is one whose output is a function of its input signals, whereas a *sequential* circuit is one whose output is determined by the order in which the input signals are applied. Sequential circuits are sometimes said to possess a sense of history. An everyday example of a combinational circuit is a domestic lighting circuit controlled by an ordinary tumbler switch. If the switch is down the light is on, and if the switch is up the light is off. A lighting circuit controlled by a cord-pull, on the other hand, is sequential, for the effect of pulling the cord depends on the current state of the circuit. If the light is on a pull turns it off, and if the light is off a pull turns it on.

Sequential circuits in turn are classified as *unclocked* (asynchronous) or *clocked* (synchronous).† Unclocked circuits are *event-driven circuits,* in contrast to clocked circuits (also known as *clock-driven circuits*), whose

† A third category of sequential circuits, referred to as *pulse-driven circuits* is discussed in the 2nd edition of 'Problems and Solutions in Logic Design', by D. Zissos, Oxford University Press, 1978.

operation is synchronized with the application of *clock pulses*, between which no changes of state can occur. In hardware terms unclocked circuits can be implemented using logic gates only, whereas the implementation of clocked circuits requires clocked flip-flops as well as gates.

Up to 1969, when the Boolean sequential equations were developed, the design of unclocked sequential circuits was achieved through an empirical choice of unrelated informal techniques paying little attention to engineering constraints until, in most cases, the implementation stage. The advent of the sequential equations has made possible the development of clear-cut step-by-step design procedures in which realistic circuit constraints are taken into account at the design stage. No engineering or other specialist knowledge is necessary to use these design procedures.

1.2 OPTIMAL DESIGN

The primary design objective is to allow the reader to produce sound and reliable designs which are meaningful not only to the designer, but also to the user. Elegance of design, while not specifically sought, can be achieved.

1.3 BOOLEAN ALGEBRA

The necessary basis for the design of logic circuits is a working knowledge of Boolean algebra.

In Boolean algebra, as in conventional algebra, we combine variables with operators into expressions. The Boolean variables may assume one of two values only, 0 or 1. These are not the 'zero' and 'one' of arithmetic. For example, the Boolean '0' does not mean 'nothing'. It can be used to indicate one of the two states of a two-state device, such as a flip-flop or a relay, while the other state will be indicated by a Boolean '1'. Although there exists a wide number of Boolean operators, such as AND, OR, NOT, INVERT, NOR, NAND, EXCLUSIVE-OR, etc., we need only consider three operators at this stage—all other operators can be expressed in terms of these. They are

Boolean addition (or disjunction);
Boolean multiplication (or conjunction);
Boolean inversion (or negation).

The addition (or disjunction) operator is written as '+'. Sometimes it is written as '$\vee$', or '$\cup$' or 'OR'. '$A+B$' may be read 'A or B' or 'A plus B'. '$A+B$' is true if either A or B or both are true, and false otherwise. Thus,

$$0+0=0$$
$$0+1=1$$
$$1+1=1$$
$$1+0=1$$

The multiplication (or conjunction) operator is written as '$\cdot$' or '$\times$'. Often it is omitted when its factors are variables denoted by single letters (the same rule as in ordinary algebra). Sometimes it is written as '$\wedge$', or '$\cap$' or 'AND'. '$A \cdot B$' may be read 'A and B', or 'A times B'. '$A \cdot B$' is true if A and B are both true, and false otherwise. Thus,

$$0\times 0=0$$
$$0\times 1=0$$
$$1\times 1=1$$
$$1\times 0=0$$

The inversion (or complementing or negation) operator is written as a bar over its argument or as a $\neg$ in front of it. Sometimes it is written as 'NOT'. Thus the inverse of A is $\bar{A}$, or $\neg A$, or 'NOT' A.

Boolean Theorems[1, 2]

For our purpose, which is to design and implement digital circuits and systems, we need only three theorems. A theorem to allow us to remove redundancies in a circuit, a theorem to suppress unwanted signal spikes (race-hazards), and De Morgan's theorem. We refer to the first two theorems as *the redundancy theorem* and *the race-hazard theorem* respectively. We list our three theorems below.

Theorem 1 *Redundancy theorem*

$$A+AB=A$$

Proof

$$\begin{aligned} A+AB &= A(1+B) \\ &= A\cdot 1 \\ &= A \end{aligned}$$

This theorem states that in a sum-of-products Boolean expression, a product that contains all the factors of another product is redundant. It allows us to eliminate redundant products in a sum-of-products expression. For example, in the Boolean function $f = AB+ABC+ABD$, the products ABC and ABD can be eliminated, because each contains all the factors present in AB.

Theorem 2 *Race-hazard theorem*

$$AB+\bar{A}C=AB+\bar{A}C+BC$$

Proof

$$\begin{aligned} AB+\bar{A}C+BC &= AB+\bar{A}C+(A+\bar{A})BC \\ &= AB+\bar{A}C+ABC+\bar{A}BC \\ &= AB(1+C)+\bar{A}C(1+B) \\ &= AB+\bar{A}C \end{aligned}$$

This theorem allows us to introduce optional† products into a sum-of-products expression. The optional product is the product of the coefficients of A and $\bar{A}$ in the expression $AB+\bar{A}C$. The product BC is optional so long as its parent products (AB and $\bar{A}C$) remain in the expression. Should, however, one of its parent products be eliminated (by applying theorem 1), then such a product is no longer optional, and cannot be removed from the expression. We shall demonstrate this property by three examples.

Example 1

In the Boolean expression $f = A+\bar{A}B$, we observe that one of the two products A, which can be written as $A\cdot 1$, contains A, and another product $\bar{A}B$ contains $\bar{A}$. Therefore, using theorem 2, we can introduce the optional product $1\cdot B = B$, thus

$$f = A+\bar{A}B+(B)$$

Now, by theorem 1, product $\bar{A}B$ is redundant, because it contains all the factors (in this case, simply B) of product B. Since $\bar{A}B$, one of the parent products of B, is not now present in the expression, the term B is no longer optional.

Diagrammatically, we show these steps as follows

$f = A+\bar{A}B$

B—replaces parent product $\bar{A}B$.

$= A+B$—**the required result.**

Example 2

Consider the Boolean expression $f = AB+\bar{A}C+BCD$. Because of the presence of A in the product AB and the presence of $\bar{A}$ in the product $\bar{A}C$, we can use theorem 2 to introduce the optional product BC. Thus,

$$f = AB+\bar{A}C+BCD+(BC)$$

Now, by theorem 1, the product BCD is redundant, since it contains all the factors of the product BC. Therefore,

$$f = AB+\bar{A}C+(BC)$$

† A Boolean product is defined as optional if its presence in an expression does not affect the value of the function, and will normally be bracketed.

Now, because the parent products of BC, namely AB and $\bar{A}C$, are still present in the expression, the term BC is redundant, and therefore it can be eliminated, leaving

$$f = AB + \bar{A}C$$

Diagrammatically, we show these steps as follows

$$f = AB + \bar{A}C + BCD$$

BC—eliminates non-parent product BCD.

$$= AB + \bar{A}C$$—**the required result.**

Example 3

Consider the Boolean expression $f = A + \bar{A}B + BC$. The optional product B, generated from the first two products A and $\bar{A}B$, replaces its parent product $\bar{A}B$ and eliminates non-parent product BC. Diagrammatically, we show this process as follows

$$f = A + \bar{A}B + BC$$

B—replaces parent product $\bar{A}B$ and eliminates non-parent product BC.

$$= A + B$$—**the required result.**

In summary, *an optional product can be used* (*i*) *to eliminate non-parent products, and/or* (*ii*) *to replace parent products.*

Theorem 3 *De Morgan's theorem*

The complement of a Boolean expression can be derived directly by replacing each variable by its complement in the corresponding dual expression. For example, the dual of $P = A + BC$ is

$$A \cdot (B + C)$$

Therefore, by De Morgan's theorem the complement of P is

$$\bar{P} = \bar{A} \cdot (\bar{B} + \bar{C})$$

Proof. If the expression to be inverted is P, and the expression resulting from replacing each variable in the dual of P by its complement is Q, we have to prove that $Q = \bar{P}$. This will be so if and only if

$$P \cdot Q = P \cdot \bar{P} = 0$$

and

$$P + Q = P + \bar{P} = 1$$

(i) Suppose P is simply a constant (0 or 1) or a variable, say $P = A$. Then $Q = \bar{A} = \bar{P}$. Further, if $P = \bar{A}$ (an inverted variable), then

$$Q = \bar{\bar{A}} = A = \bar{P}$$

(ii) Suppose P is a sum of two terms A, B (which may well be expressions). Then

$$P = A + B$$
$$Q = \bar{A}\bar{B}$$

Therefore,

$$PQ = (A+B)\cdot\bar{A}\bar{B} = A\bar{A}\bar{B} + \bar{A}\bar{B}B = 0+0=0$$

and

$$\begin{aligned} P+Q &= A+B+\bar{A}\bar{B} = A+B+\bar{A}\bar{B}+\bar{B} && \text{(Theorem 2)} \\ &= A+B+\bar{B} && \text{(Theorem 1)} \\ &= A+1 \\ &= 1 \end{aligned}$$

Therefore $Q = \bar{P}$.

(iii) Suppose P is a product of two factors A, B (which may well be expressions).
Then

$$P = AB$$
$$Q = \bar{A} + \bar{B}$$

Therefore,

$$\begin{aligned} PQ &= AB(\bar{A}+\bar{B}) \\ &= 0+0 = 0 \end{aligned}$$

and

$$P+Q = AB+\bar{A}+\bar{B} = AB+\bar{A}+\bar{B}+B=1$$

Therefore $Q = \bar{P}$.

Before inverting a given expression it is advisable (a) to simplify the expression and (b) to include all product terms in brackets. The brackets remain unaffected by the complementing process.

Example 1

Derive the complement of $P = A + B\bar{C}$.

Suggested procedure

Given	$P = A + B\bar{C}$
Minimize	$P = A + B\bar{C}$

Bracket all products	$P = A+(B\bar{C})$
Invert	$\bar{P} = \bar{A}\cdot(\bar{B}+C)$
Remove redundant brackets	$\bar{P} = \bar{A}\cdot(\bar{B}+C).$

Example 2

Derive the complement of $f = A(BC+\bar{B}\bar{C}+BCD)$.

Suggested procedure

Given	$f = A(BC+\bar{B}\bar{C}+BCD)$
Minimize	$f = A(BC+\bar{B}\bar{C})$
Bracket all products	$f = A[(BC)+(\bar{B}\bar{C})]$
Invert	$\bar{f} = \bar{A}+[(\bar{B}+\bar{C})\ (B+C)]$
Remove redundant brackets	$\bar{f} = \bar{A}+(\bar{B}+\bar{C})\ (B+C).$

Example 3 (The gossip problem)

Given that

(1) Alice never gossips,

(2) Betty gossips if and only if Alice is present,

(3) Clarice gossips under all conditions even when alone,

(4) Dorothy gossips if and only if Alice is present.

Determine the conditions when there is no gossip in the room.

SOLUTION

Let $G = 1$ indicate that there is gossip in the room; thus $G = 0$ indicates that there is no gossip. Let $A = 1$ indicate the presence, and $A = 0$ the absence, of Alice. Similarly let B, C, D refer to Betty, Clarice, and Dorothy respectively. Translating the given conditions into a Boolean equation we have

$$G = AB+C+AD$$

(the terms are respectively the given conditions (2), (3), (4); condition (1) contributes the term $A\cdot 0$, which is 0).

To derive $\bar{G}$ (the condition for no gossip) we proceed conventionally.

Given	$G = AB+C+AD$
Minimize	$G = AB+C+AD$
Bracket all products	$G = (AB)+C+(AD)$
Invert	$\bar{G} = (\bar{A}+\bar{B})\bar{C}(\bar{A}+\bar{D})$
Remove redundant brackets	$\bar{G} = \bar{A}\bar{C}+\bar{B}\bar{C}\bar{D}.$

Therefore there is no gossip if both Alice and Clarice are absent or Betty, Clarice and Dorothy are all absent.

Before inverting a given expression it is advisable (a) to reduce the expression and (b) to include all product terms in brackets. The brackets remain unaffected by the complementing process.

Example

Derive the complement of $P = A + B\bar{C} + AD$

Suggested procedure

Given	$P = A + B\bar{C} + AD$
Reduce	$P = A + B\bar{C}$
Bracket all products	$P = (A) + (B\bar{C})$
Invert	$\bar{P} = (\bar{A}) \cdot (\bar{B} + C)$
Remove redundant brackets	$\bar{P} = \bar{A} \cdot (\bar{B} + C)$—**the required result.**

Boolean Reduction

A Boolean function is said to be *irredundant*, or *reduced*, if it contains no optional products or factors, that is products or factors whose presence does not affect the value of the function. For example, factor $\bar{A}$ in $A + \bar{A}B$ is redundant, since $A + \bar{A}B = A + B$. Redundancies in two-level Boolean expressions can be removed in three steps, using theorems 1 and 2. If an expression contains more than two levels, such as $f = A + B(C + D)$, we convert it into its two-level sum-of-products form by multiplying out.

The three steps for eliminating redundancies in Boolean expressions are as follows

Step 1 *Multiply out*

The expression to be reduced is converted into its two-level sum-of-products form by multiplying out. Products that contain both a variable and its complement as factors are eliminated, using the identity $A \cdot \bar{A} = 0$. The repetition of a variable in a product is eliminated using the identity $A \cdot A = A$. The products are finally re-arranged in ascending order of size from left to right.

Example

Consider the Boolean function $f = BC + (AB + D)\bar{D} + A$. Applying step 1, we obtain

$$\begin{aligned} f &= BC + (AB + D)\bar{D} + A \\ &= BC + AB\bar{D} + D\bar{D} + A \\ &= BC + AB\bar{D} + A \\ &= A + BC + AB\bar{D} \end{aligned}$$

Step 2 *Apply theorem 1*

We eliminate redundant products using theorem 1, as follows. Starting with the products of the fewest factors, that is from the left, we take each term in succession and compare with it all products containing more factors; these will be to its right. A product that contains all the factors of the given term is eliminated.

Example

In step 1, we derived $f = A + BC + AB\bar{D}$. We start step 2 by considering the first product, in this case A. We scan the products to the right of A, looking for a product that contains A as a factor. $AB\bar{D}$ is such a product, which therefore is eliminated, resulting in $f = A + BC$. Since there are no products to the right of BC, we do not repeat the step.

Step 3 *Apply theorem 2*

Here we generate optional products, using theorem 2. In practice, we find that experience will enable us to take short cuts in the process described below. However, a complete systematic description is given for use by beginners or in a computer program.

Assuming the products are arranged in ascending order of size from left to right, we proceed as follows.

(1) The first variable in the first product is selected, and the remainder of the expression is scanned for a product that contains the complement of the selected variable. When such a product is found, we form an optional product, using theorem 2. The optional product is used to eliminate non-parent products and/or to replace parent products, as illustrated in Example 3 following theorem 2. If a parent product has been replaced, we insert the optional product at the beginning of the expression and we repeat step 3. If the optional product has not been used, it is discarded.

Step 3 is repeated until all first-level optional products have been generated.

(2) We repeat step 3, using higher-level optional products. We shall demonstrate the reduction steps by means of the following examples.

Example 1

Reduce $f = A + \bar{A}\bar{B} + \bar{B}DC + \bar{A}BD$.

SOLUTION

Step 1 *Multiply-out*

No change

Step 2 *Apply theorem 1*

No change

Step 3 *Apply theorem* 2

$$f = A + \bar{A}\bar{B} + \bar{B}CD + \bar{A}BD$$

$\bar{B}$ replaces parent product $\bar{A}\bar{B}$ and eliminates non-parent product $\bar{B}CD$

$$= A + \bar{B} + \bar{A}BD$$

BD replaces parent product $\bar{A}BD$

$$= A + \bar{B} + BD$$

D replaces parent product BD

$$= A + \bar{B} + D$$—**the required result.**

Example 2

Reduce $f = (A + C)(\bar{A} + B) + DE[\bar{B} + \bar{C}] + ABC$

SOLUTION

Step 1 *Multiply-out*

$$f = A\bar{A} + AB + \bar{A}C + BC + \bar{B}DE + \bar{C}DE + ABC$$
$$= AB + \bar{A}C + BC + \bar{B}DE + \bar{C}DE + ABC$$

Step 2 *Apply theorem* 1

$$f = AB + \bar{A}C + BC + \bar{B}DE + \bar{C}DE$$

ABC is redundant because it contains all the factors of product AB(A and B).

Step 3 *Apply theorem* 2

$$f = AB + \bar{A}C + BC + \bar{B}DE + \bar{C}DE$$

BC—eliminates non-parent product BC

$$= AB + \bar{A}C + \bar{B}DE + \bar{C}DE$$

BC

CDE

DE—replaces parent products $\bar{B}DE$ and $\bar{C}DE$

$$= AB + \bar{A}C + DE$$—**the required result.**

1.4 GATES

NAND Gates

Although logic circuits can be constructed using a mixture of AND, OR, INVERTER, NOR, NAND, EXCLUSIVE—OR ... gates, for the sake of simplicity we shall only use NAND gates, flip-flops and tristates. It must however be stressed that our methods allow any type of logic element to be used.

A NAND gate generates the OR function of the inverted inputs. For example, if signals A, $\bar{B}$ and C are fed into a NAND gate, its output is $\overline{A\bar{B}C} = \bar{A}+B+\bar{C}$—see Figure 1.1(a).

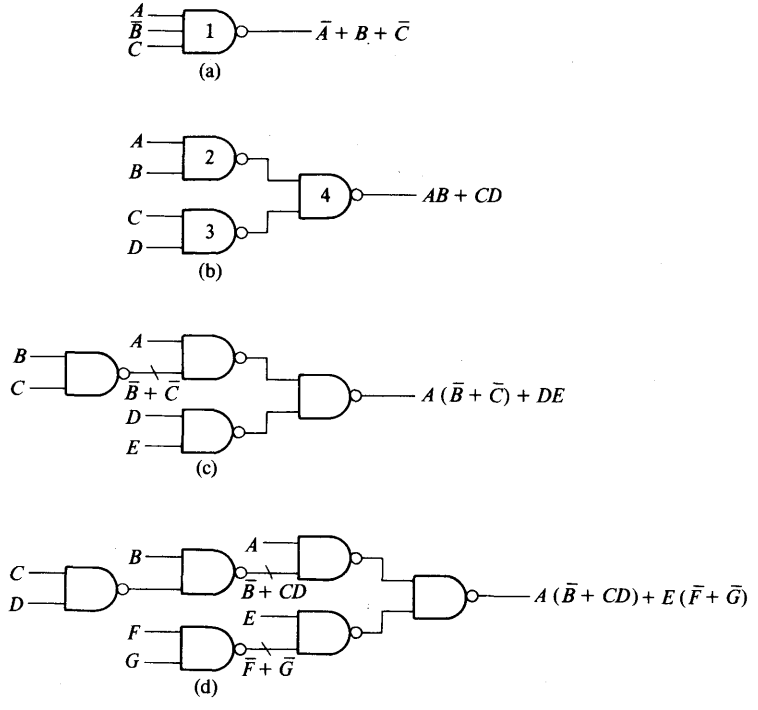


Figure 1.1

The output of a NAND circuit can be expressed as a Boolean sum-of-products expression, one product for each gate driving the output gate. The factors of each are the input signals to the corresponding gate. For example the

output of circuit 1.1(b) is $AB+CD$. That this is so can be shown as follows

$$g_4 = \overline{g_2 \cdot g_3}$$
$$= \bar{g}_2 + \bar{g}_3$$

Now,

$$g_2 = \bar{A} + \bar{B}, \text{ and}$$
$$g_3 = \bar{C} + \bar{D}$$

Therefore,

$$g_4 = \overline{\bar{A} + \bar{B}} + \overline{\bar{C} + \bar{D}}$$
$$= AB + CD$$

The inexperienced reader is advised to use these steps to derive the outputs of the circuits in Figures 1.1(c) and 1.1(d).

The same steps in the reverse order can be used to implement Boolean functions with NAND gates. For example, to implement the function $f = B\bar{C} + A(B+\bar{D})$, we proceed as follows.

1. We first draw the output NAND gate—gate 1 in Figure 1.2.
2. Next we draw two NAND gates, one for each of the two products $B\bar{C}$ and $A(B+\bar{D})$.
3. The input signals of gate 2 are B and $\bar{C}$, and of gate 3, A and $B+\bar{D}$.
4. Finally we generate signal $B+\bar{D}$. For that we need a fourth gate the inputs of which are $\bar{B}$ and D, as shown in Figure 1.2.

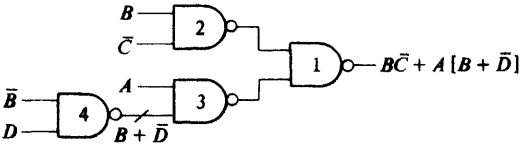


Figure 1.2.

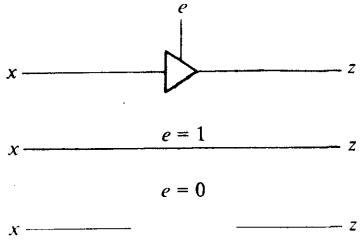


Figure 1.3.

Tristates

Tristates were developed in 1969 by H. Mine and others at Kyoto University. Each gate has one input, one output and one enable terminal, as shown in Figure 1.3. When $e = 1$ the gate behaves like a short circuit, that is the output follows the input $z = x$. When $e = 0$ the gate is tristated, that is it behaves like an open circuit.

1.5 RACE HAZARDS [3]

Race hazards are unwanted transient signals (signal spikes) which, under certain changes of an input signal and with certain relationships of circuit delays, appear in a logic circuit. Figure 1.4 shows an example in which 'spikes' occur during a change of input signal A from 1 to 0 when $B = C = 1$. The cause of race hazards is that immediately following a change in a signal A, $A = \bar{A}$ = either 0 or 1. It follows that if the Boolean expression of a signal in a circuit reduces to either of the two forms $A + \bar{A}$ or $A \cdot \bar{A}$, a race hazard exists at the output of the corresponding gate—otherwise the signal is hazard-free.

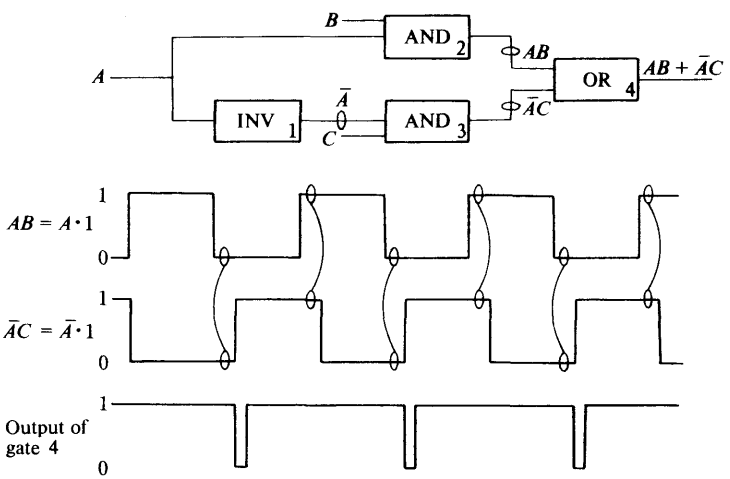


Figure 1.4.

Returning to our example in Fig. 1.4, $f = AB + \bar{A}C$ which reduces to $A + \bar{A}$ when $B = C = 1$, revealing the existence of a race hazard at the output of gate 4. Race hazards in a circuit clearly can be suppressed by preventing its Boolean expression from reducing to either of the two forms $A + \bar{A}$ or $A \cdot \bar{A}$.

This is readily achieved by means of theorem 2, namely

$$AB+\bar{A}C = AB+\bar{A}C+BC$$

or

$$(A+B)(\bar{A}+C) = (A+B)(\bar{A}+C)(B+C)$$

The introduction of the third term prevents the first expression from being reduced to $A+\bar{A}$, since when $B = C = 1$, $AB+\bar{A}C+BC$ reduces to $A+\bar{A}+1 = 1$. Similarly when $B = C = 0$, the second expression reduces to $(A+0)(\bar{A}+0)(0+0) = A \cdot \bar{A} \cdot 0 = 0$.

Race-hazards are automatically eliminated in sequential circuits which are designed using our steps and are implemented with gates of maximum speed tolerance of $\pm 33\frac{1}{3}\%$. The design steps are described in section 1.9 and the $33\frac{1}{3}\%$ property is proved in section 1.8.

1.6 UNUSED STATES

If the number of states of a logic circuit to be implemented is N, where $2^{n-1} < N < 2^n$, there will be $2^n - N$ *unused states*. The reader is strongly advised against using these states as 'don't care' circuit conditions. This is because in practice we cannot ignore the possibility of a circuit assuming an unused state. The designer must therefore take such a possibility into account at the design level and specify the desired action. This means that all state diagrams must have 2^n states before they are implemented.

1.7 STATE REDUCTION

Under certain conditions it is possible to reduce the number of states in a state diagram. The conditions have been defined by Caldwell[4] and are listed below.

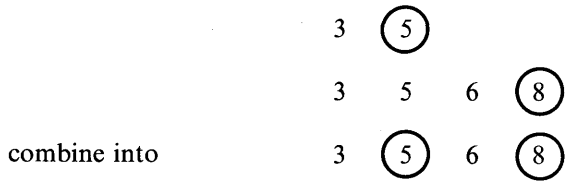
The state diagram is translated into a state table which has as many rows as states and as many columns as combinations of input signals (input states). Each row corresponds to a state in the diagram, and each column to an input state. The rows and columns are headed by labels representing the corresponding inputs and states. In each square we enter the circuit destination, that is, the next state that the circuit assumes when it is in a state represented by the row heading, and the input signals are those specified by the column heading.† If the designer does not wish to specify the next state to be

† In the case of clocked circuits, we omit the clock signals from our state tables since it has already been specified that circuit changes can only be initiated by clock pulses.

assumed under certain conditions, he can leave the entry in the corresponding square blank. As in the case of state diagrams, in each square we must specify the circuit outputs, unless it is a blank square. Clearly, if the circuit destination is the same as its current state, the circuit is stable—in such cases it is the convention to circle the entries.

The process of combining the rows of a state table is made in accordance with the following rules.

1. Two rows may be merged if the state numbers and the circuit outputs appearing in corresponding columns of each row are alike, or if the entry in one or both of the rows is blank.
2. When circled and uncircled entries of the same state number are to be combined, the resulting entry is circled. Thus the two rows



Note that a change from state 5 to state 8 now involves a change of the input state only. When a row S_m is merged with a row S_n we shall denote the new row by S_{mn}.

See pages 23 and 28 for applications of these steps. For additional examples see [1].

1.8 SEQUENTIAL EQUATIONS

The operation of unclocked sequential circuits can be expressed algebraically by means of Boolean statements, commonly referred to as *sequential equations* [1, 2]. It is the development of these equations in 1969 that has made possible in turn the development of clear-cut step-by-step procedures for logic circuits in which circuit constraints are taken into account at the design level.

There are two basic forms of sequential equations. They are

$$A = \Sigma \text{ turn-on sets of } A + A \cdot \overline{\Sigma \text{ turn-off sets of } A} \tag{1.1}$$

$$\mathrm{A} = [\Sigma \text{ turn-on sets of A} + \mathrm{A}] \cdot \overline{\Sigma \text{ turn-off sets of A}} \tag{1.2}$$

the terms having meaning as follows.

Variable A is a secondary signal (state variable).

Turn-on set of a secondary signal is a set of Boolean variables, which when

equal to 1, cause the secondary signal to turn on (that is to assume the value of 1). By analogy,

Turn-off set of a secondary signal is a set of Boolean variables, which when equal to 1, cause the secondary signal to turn off (that is to assume the value of 0).

We refer to these equations as *primitive sequential equations* and to their direct circuit implementations as *primitive sequential circuits.*

Equation 1.1 is used when the design is to be implemented with NAND gates and equation 1.2 when it is to be implemented with NOR gates. We therefore refer to them as *NAND and NOR sequential equations* respectively.

The application of the sequential equations is not confined to NOR and NAND gates, but can be extended to all types of digital elements, such as electromechanical relays, fluidic gates and so on. In Figures 1.5(a) and 1.5(b) we show the relay implementations of the NOR and NAND sequential equations repectively. 'Push-to-make' switch s generates the turn-on set of relay A and 'push-to-break' switch $\bar{r}$ its turn-off set.

Note that if due to some adverse circuit condition the turn-on and turn-off sets of a secondary signal are present at the same time, in the case of the NAND equation the turn-on set will override the turn-off set. The reverse is true for the NOR equation. This property may be used when designing fail-safe systems.

The turn-on and turn-off sets of secondary signals are derived directly from the state diagram. For example, from Figure 1.6(a)† we obtain

$$\text{turn-on set of } A = B\bar{c}$$

$$\text{turn-off set of } A = \bar{B}\bar{c}$$

$$\text{turn-on set of } B = \bar{A}c$$

$$\text{turn-off set of } B = Ac$$

Substituting these values in equation 1.1 we obtain the circuit's NAND equations. They are

$$A = B\bar{c} + A(B + c) \tag{1.3}$$

$$B = \bar{A}c + B(\bar{A} + \bar{c}) \tag{1.4}$$

$$Z = S2 + S3 = AB + A\bar{B} = A$$

The corresponding NAND circuit is shown in Figure 1.6(b).

The gate count can be minimized by applying to the equations the processes of *merging* and *signal substitution.* Although these processes are formalized, their application introduces obscurities in the circuit and affects the relative

† This is the internal state diagram of a master-slave TFF (T flip-flop).

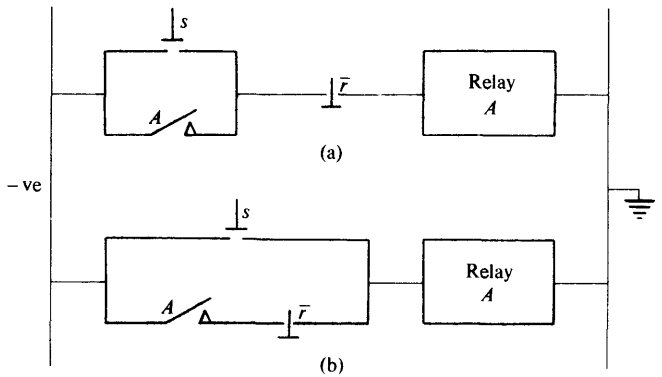


Figure 1.5.

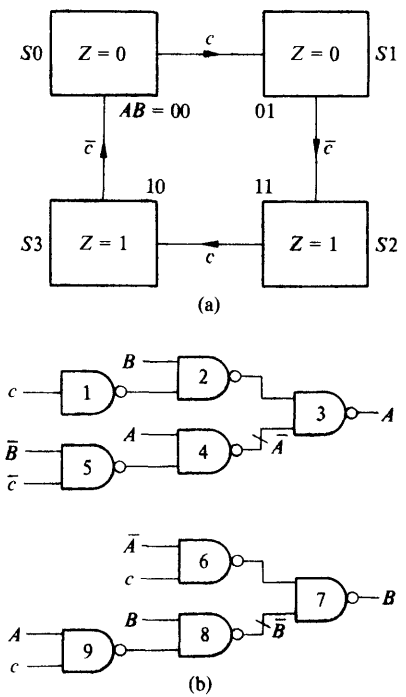


Figure 1.6.

signal delays. For this reason we shall not use them. The interested reader is referred to [2].

Inverted Signal

That the outputs of gates 4 and 8 in Figure 1.6 are $\bar{A}$ and $\bar{B}$ can be proved as follows. Let us denote by s the Σ turn-on sets of M, where M is a secondary signal, and by r the Σ turn-off sets of M. Then

$$M = s + M \cdot \bar{r}$$

Its NAND implementation is shown in Figure 1.7.

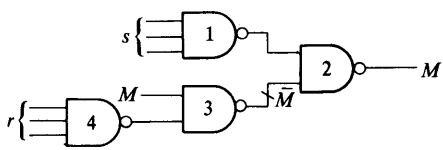


Figure 1.7.

To obtain $\overline{M}$ we invert both sides of the equation.

$$\overline{M} = \bar{s}(\overline{M} + r)$$

Adding $s \cdot \overline{M}$ and $s \cdot r$ as optional products to the right hand side of the equation, we obtain

$$\begin{aligned} \overline{M} &= \bar{s}\overline{M} + \bar{s}r + (s\overline{M}) + (sr) \\ &= \overline{M}(\bar{s} + s) + r(\bar{s} + s) \\ &= \overline{M} + r \\ &= \text{output of gate 3 in Figure 1.7.} \end{aligned}$$

Product $s\overline{M}$ can be used as optional, because when $s = 1, M$ will not equal 0. Similarly, s and r cannot equal 1 simultaneously as the turn-on and turn-off set of a secondary signal in a system are not normally generated at the same time.

Signal Delays[5]

Signal delays in logic circuits can be derived by reference to the circuit diagram or directly from the circuit's Boolean equations, as we demonstrate below. We use t_g to denote the nominal propagation time of a gate.

Referring to the circuit diagram in Figure 1.6 (b), we obtain

time for A to turn on $= 3t_g$—gates 1, 2 and 3

time for A to turn off $= 4t_g$—gates 1, 5, 4 and 3

time for B to turn on $= 2t_g$—gates 6 and 7

time for B to turn off $= 3t_g$—gates 9, 8 and 7.

When referring to the circuit's sequential equations, we proceed as follows. For ease of reference we repeat the equations below.

$$A = B\bar{c} + A[B + c]$$

$$B = \bar{A}c + B[\bar{A} + \bar{c}]$$

Signal A turns on when its turn-on set, $B\bar{c}$, becomes 1; that is when $B = 1$ and $\bar{c}$ changes to 1. The time interval between the change in c and the change in A is $3t_g$s. This is because the change of signal c is first inverted, then ANDed with B and finally ORed with $A[B+c]$ before it causes A to change to 1. Similarly,

time for A to turn off $= 4t_g$—INV, OR, AND, OR

time for B to turn on $= 2t_g$—AND, OR

time for B to turn off $= 3t_g$—OR, AND, OR.

Because of the format of our sequential equations, a change in an input signal has to propagate through at least an AND level and an OR level before the secondary signal changes. That is, the fastest time in which a secondary signal (state variable) can change is $2t_g$. Now the maximum time by which a primary signal can be delayed is t_g, when it is inverted. Allowing for $x\%$ maximum variation in the propagation time of gates caused by such factors as production spread, varying loads, ageing etc., we have

$$t_{p\,\max} = t_g(1+x), \qquad \text{and}$$

$$t_{s\,\min} = 2t_g(1-x)$$

It has been shown [1, 2, 3] that circuit misoperation is avoided if

$$t_{p\,\max} \leq t_{s\,\min}.$$

Therefore in primitive circuits,

$$t_g(1+x) \leq 2t_g(1-x)$$

$$1+x \leq 2-2x$$

$$x \leq \tfrac{1}{3}$$

That is, all our circuits are hazard-free when implemented with gates of maximum gate speed tolerance of $\pm 33\frac{1}{3}\%$.

This figure is the theoretical maximum. In practice it can be increased by allowing for the probability of the slowest gate in a circuit racing in a critical race the two fastest gates. The figure can be further increased if the filtering effect of gates is taken into account.

The reader's attention is drawn to the fact that algebraic manipulation of the sequential equations must be avoided, unless account is taken of the fact that each algebraic manipulation affects the relative delays of the primary and secondary signals. If circuit minimality is necessary or desirable the designer should apply the steps of *merging* and *signal substitution* [2].

1.9 EVENT-DRIVEN SEQUENTIAL CIRCUITS†

In this section we shall consider the step-by-step design of event-driven sequential circuits.

Design Factors

Our design process is accomplished in four steps, and meets the following design factors.

1. *Circuit reliability.* All circuits function correctly and reliably.
2. *Gate minimality.* Generally speaking not all our circuits will be minimal.
3. *Gate speed tolerance.* Variations of $\pm 33\frac{1}{3}\%$ in the response times of gates are automatically met.
4. *Circuit maintainability.* Our circuits are easy to maintain.
5. *Design effort.* This is minimal.
6. *Documentation.* No additional documentation is needed.
7. *The design steps.* These are easy to apply. No specialist knowledge of electronics is necessary.
8. *Gate fan-in and fan-out restrictions.* These are met reliably, though not elegantly. For elegance of design the interested reader is referred to [2].

Design Steps

The sequence in which the four design steps are executed with a detailed description of each step is given below.

† These circuits are also referred to as unclocked or asynchronous sequential circuits.

Step 1 *I/O characteristics*

In this step we draw a block diagram to show the available input signals and the required output signals. We next use a state diagram to define the relationship which must be established by our circuit between the two sets of signals.

Step 2 *Internal characteristics*

In the second step the designer specifies the internal performance of the circuit. Although experience, intuition and foresight play an important part at this stage, the inexperienced designer should be primarily concerned that his specification of the internal circuit operation is complete and free from ambiguities. To this end he should avoid short cuts, and should use as many states as he finds necessary to give a complete and unambiguous specification of the circuit performance. The next step can be used to eliminate unwanted states.

Step 3 *State reduction*

This step is optional and can be omitted. Its main purpose is to provide the designer with the means for reducing the number of internal states he used in step 2, if such a reduction is possible and desirable.

The circuit's state table is drawn and the state reduction steps are used to merge its rows.

Clearly to avoid redundant states we would only use this step to reduce the number of states to some power of 2. For example, whereas we would use it to reduce five states to four, we would not use it to reduce four states to three.

Step 4 *Circuit implementation*

In this step we give each internal state a unique binary code. The coding must be such that a circuit transition between two adjacent states involves the change of one secondary signal only. The race-free diagrams in Figure 1.8 can be used for this purpose.

Having coded the internal states we proceed to derive the Boolean equations for the state variables and the output signals. Although initially blank entries can exist in a state table, clearly after the circuit equations have been derived the designer must fill in the blank squares. He does so according to the use he made of the optional products defining unspecified circuit conditions. In other words no blank entries must exist in a finalized circuit design.

A Design Problem *An alarm circuit*

Design an alarm circuit with the following terminal characteristics. The appearance of a fault signal f activates an alarm bell, turns a green light off and

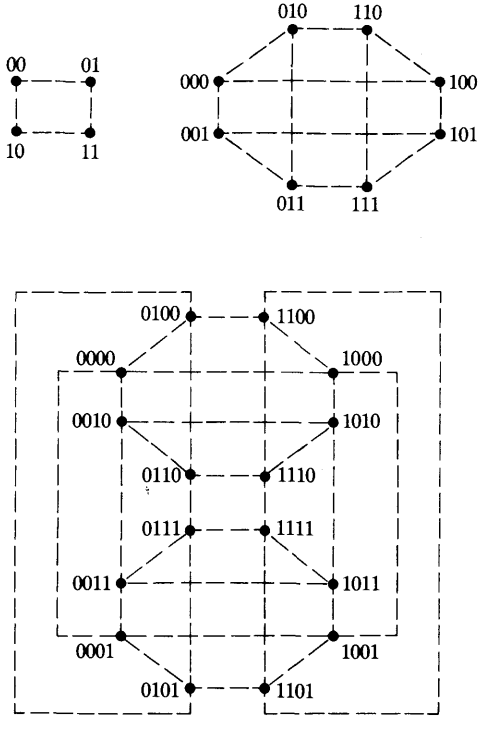


Figure 1.8.

a red light on. The operator turns off the bell by pressing an acknowledge switch *a*. When the fault clears itself, the red light turns off, the green light turns on and the bell is automatically reactivated to attract the operator's attention. The bell is turned off when the operator presses the acknowledge button.

Should the fault disappear before it is acknowledged the circuit is to assume its previous state. For further problems see [2].

SOLUTION

Step 1 *I/O characteristics*

The I/O signals are shown in the block diagram in Figure 1.9(a). The specified interplay between input and output signals is expressed by means of the state diagram in Figure 1.9(b).

Step 2 *Internal characteristics*

In this case the internal characteristics are the same as the external.

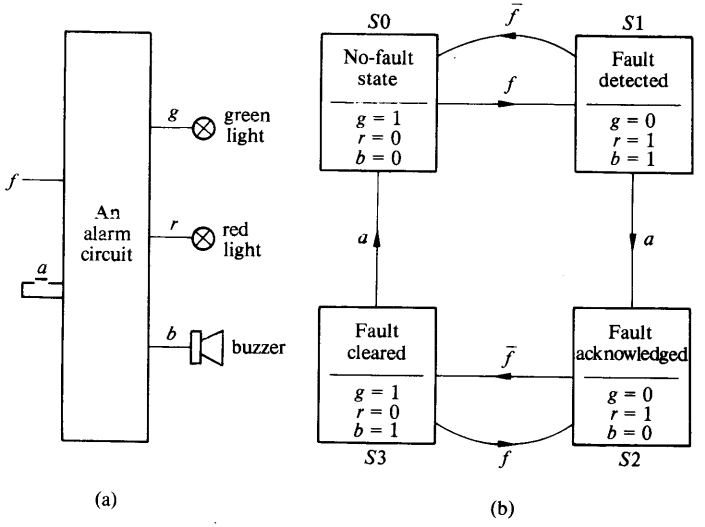


Figure 1.9.

Step 3 *State reduction*

In this step we first derive the state table that corresponds to the diagram in Figure 1.9(b). This is shown in Figure 1.10(a). Because the circuit outputs are two lights and one buzzer, which would not respond to narrow signal spikes at their input, the designer has the option of defining them either as 0^s or 1^s during a circuit transition, that is in squares in the state table with uncircled entries. Next we apply the state reduction steps outlined in section 1.8. Rows *S*0, *S*1 and *S*2, *S*3 merge into rows *S*01 and *S*23 respectively, reducing our four-row table to the two-row table shown in Figure 1.10(b). The corresponding state diagram is shown in Figure 1.11(a). Using the 0 entries in our two-state table as optional products, we obtain

In state *S*01,

$$g = \bar{a}\bar{f} + a\bar{f} + (af) = \bar{f}, \quad \text{that is } g = 0 \text{ in square 3.}$$

$$r = \bar{a}f + (af) = f, \quad \text{that is } r = 1 \text{ in square 3.}$$

$$b = \bar{a}f + (af) = f, \quad \text{that is } b = 1 \text{ in square 3.}$$

In state S23,

$$g = \bar{a}\bar{f} + (a\bar{f}) = \bar{f}, \quad \text{that is } g = 1 \text{ in square 8.}$$

$$r = \bar{a}f + af + (a\bar{f}) = f, \quad \text{that is } r = 0 \text{ in square 8.}$$

$$b = \bar{a}\bar{f} + (a\bar{f}) = \bar{f}, \quad \text{that is } b = 1 \text{ in square 8.}$$

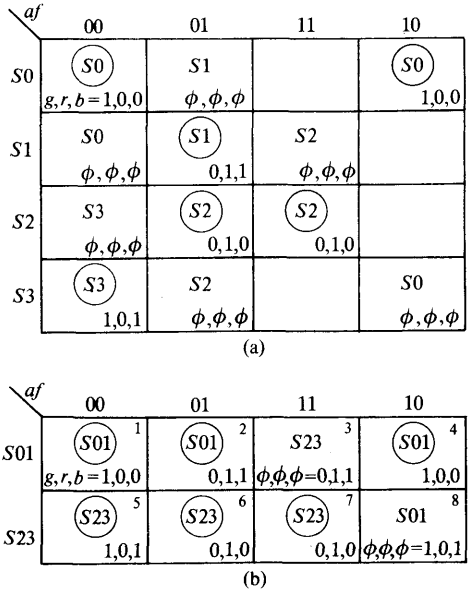


Figure 1.10.

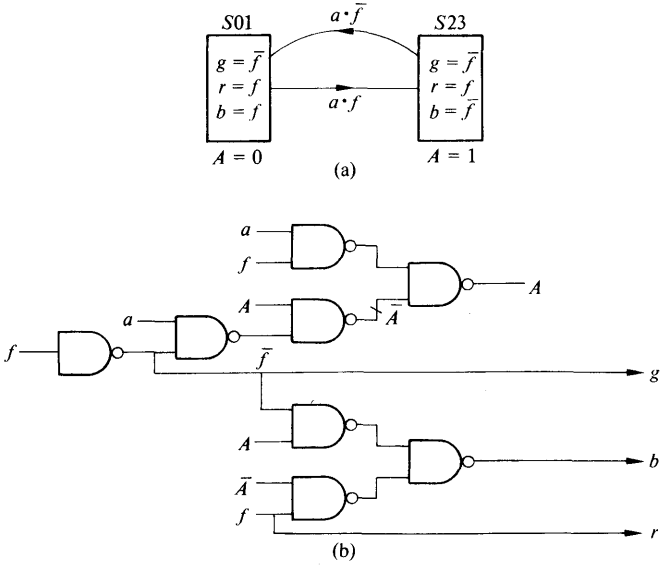


Figure 1.11.

Step 4 *Circuit implementations*

By direct reference to Figure 1.11(a), we obtain

$$\text{turn-on set of } A = a{\cdot}f$$

$$\text{turn-off set of } A = a{\cdot}\bar{f}$$

Therefore

$$A = af + A(\bar{a} + f)$$

$$g = S01{\cdot}\bar{f} + S23{\cdot}\bar{f} = \bar{A}\bar{f} + A\bar{f} = \bar{f}$$

$$r = S01{\cdot}f + S23{\cdot}f = \bar{A}f + Af = f$$

$$b = S01{\cdot}f + S23{\cdot}\bar{f} = \bar{A}f + A\bar{f}$$

The corresponding NAND circuit is shown in Figure 1.11(b).

1.10 CLOCK-DRIVEN SEQUENTIAL CIRCUITS†

Functionally, the essential characteristics of clock-driven circuits are

(a) their operation is synchronized with the application of clock pulses, between which no changes of state can occur, and
(b) any number of state variables can change during a circuit transition.

In hardware terms, the state variables are produced by means of clocked flip-flops. These are bistable elements in which the change of the output signal A is coincident with either the leading or the trailing edge of a pulse signal, commonly referred to as the *clock pulse*. Throughout this book, unless we specify otherwise, it will be assumed that a change in the output signal, A, takes place on the trailing edge of the clock pulse.

There are four basic types of flip-flops, namely

(i) D flip-flops (DFFs)
(ii) T flip-flops (TFFs)
(iii) SR flip-flops (SRFFs), and
(iv) JK flip-flops (JKFFs).

Their terminal characteristics are shown in Figure 1.12. Their implementation is discussed in Chapter 2 of Zissos, D., 'Problems and Solutions in Logic Design', Oxford University Press, 1976.

Their terminal characteristics are shown in Figure 1.12.

† These circuits are also referred to as clocked or synchronous sequential circuits.

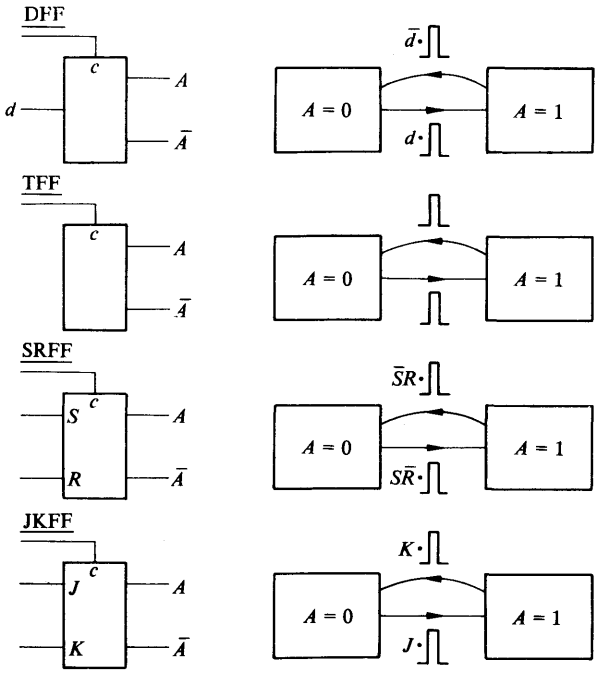


Figure 1.12.

The design of clock-driven circuits is accomplished in four steps. These steps are identical to those used in the design of event-driven circuits with the following exceptions. The state variables are defined by flip-flop equations, in contrast to sequential equations used in event-driven circuits. The flip-flop equations are Boolean expressions defining the turn-on and turn-off conditions of the circuit flip-flops. The turn-on conditions of SRFF, denoted by S_A, is the disjunction (ORing) of the total states† which are necessary to cause A to change value from 0 to 1. Similarly the turn-off condition of A, denoted by R_A, is the disjunction of the total states, which are necessary to cause A to change value from 1 to 0.

The expressions for the turn-on and turn-off conditions of a flip-flop, can be reduced using as optional products

(a) products defining unspecified circuit conditions,
(b) products that allow the turn-on condition of a flip-flop to arise during a transition in which the flip-flop output remains static at 1, and

† A total state is a state which is defined by a unique combination of input and secondary signals.

(c) products that allow the turn-off condition of a flip-flop to arise during a transition in which the flip-flop output remains static at 0.

The turn-on and turn-off conditions, as derived by the foregoing process, define directly the set and reset signals for SRFF's. However in practice one uses JKFF's as they are more versatile and readily available. To obtain the equations for the J and K signals we drop the $\bar{A}$ and A variables from the equations defining S_A and R_A. The most straightforward method to prove this is by implementing the JKFF characteristics using an SR flip-flop. In Figures 1.13(a) and (b) we show the block diagram and I/O characteristics of the JKFF. The internal characteristics are the same as the external. Therefore by direct reference to Figure 1.13(b) we obtain

$$S_A = S0 \cdot J = \bar{A} \cdot J$$

$$R_A = S1 \cdot K = A \cdot K$$

The corresponding circuit is shown in Figure 1.13(c).

We shall demonstrate the steps by means of a design problem. For further problems see [2].

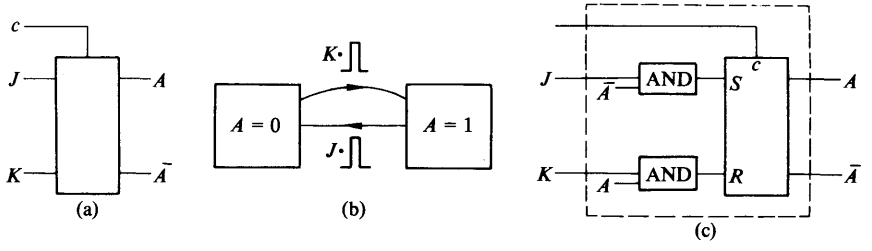


Figure 1.13.

A Design Problem *4–5–6 Detector*

Design a circuit that will stop the paper-tape reader shown in Figure 1.14 (by turning signal m off) and turn on a buzzer when the character sequence 4-5-6 is detected.

A synchronizing pulse is generated by the reader on line s each time a new character is output.

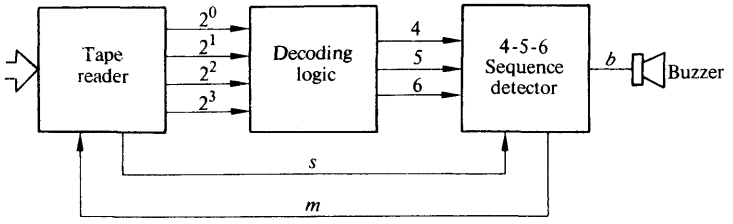


Figure 1.14.

SOLUTION

Step 1 *I/O characteristics*

As stated.

Step 2 *Internal characteristics*

A suitable internal state diagram is shown in Figure 1.15(a). In its initial state, *S*0, the circuit looks for a '4' and ignores all other characters. This is implemented by causing a circuit transition to a new state, in our case state S1, when a '4' is detected and specifying no response for all other characters.

In state *S*1 our circuit looks for a '5', the second character in our sequence. When it sees a '5' it moves to state *S*2. If a '4' is detected the circuit does not change state, allowing for the possibility of 4_s preceding our sought sequence. An input other than 4 or 5, that is $\bar{4}\cdot\bar{5}$, resets the circuit to its initial state by causing an *S*1 to *S*0 transition.

When in state *S*2 our circuit looks for a '6'. When it detects a '6' it moves to state *S*3, where the reader is turned off and the buzzer turned on. A '4' initiates transition to state *S*1. All other characters, that is $\bar{4}\cdot\bar{6}$ reset the circuit.

Step 3 *State reduction*

The corresponding state table is shown in Figure 1.15(b). No merging of rows is possible.

Step 4 *Circuit implementation*

Arbitrarily chosen codes for our four states are shown in Figure 1.15(a). In order to accommodate the reader who has little exposure to Boolean Algebra, we shall not make use of optional products to minimize the circuit implementation.

By direct reference to the state diagram in Figure 1.15(a), we obtain

$$S_A = S1\cdot 5$$
$$= \bar{A}\cdot B\cdot 5, \qquad \text{therefore } J_A = B\cdot 5$$

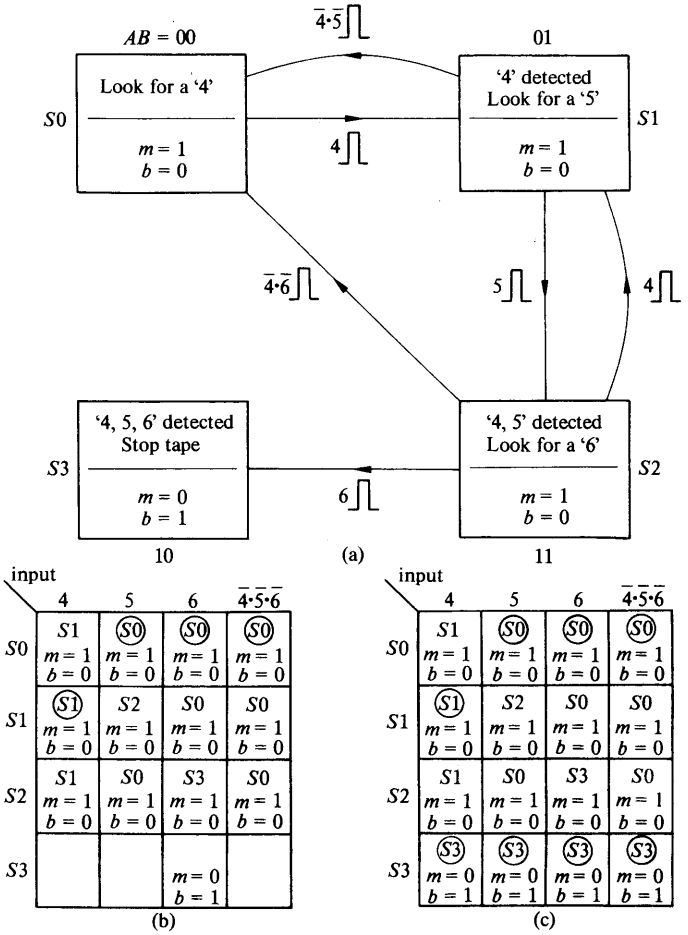


Figure 1.15.

$$R_A = S2 \cdot 4 + S2 \cdot \bar{4} \cdot \bar{6}$$

$$= S2 \cdot 4 + S2 \cdot \bar{6}$$

$$= S2 \cdot \bar{6}, \text{ since } S2 \cdot 4 \text{ is a subset of } S2 \cdot \bar{6}$$

$$= A \cdot B \cdot \bar{6}, \qquad \text{therefore } K_A = B \cdot \bar{6}$$

$$S_B = S0 \cdot 4$$

$$= \bar{A} \cdot \bar{B} \cdot 4, \qquad \text{therefore } J_B = \bar{A} \cdot 4$$

$$R_B = S1 \cdot \bar{4} \cdot \bar{5} + S2 \cdot \bar{4} \cdot \bar{6} + S2 \cdot 6$$
$$= S1 \cdot \bar{4} \cdot \bar{5} + S2 \cdot \bar{4} + S2 \cdot 6$$
$$= S1 \cdot \bar{4} \cdot \bar{5} + S2 \cdot \bar{4}, \text{ since } S2 \cdot 6 \text{ is a subset of } S2 \cdot \bar{4}$$
$$= \bar{A} \cdot B \cdot \bar{4} \cdot \bar{5} + A \cdot B \cdot \bar{4}$$
$$= B \cdot \bar{4} \cdot \bar{5} + A \cdot B \cdot \bar{4}, \qquad \text{therefore } K_B = \bar{4} \cdot \bar{5} + A \cdot \bar{4}$$

$$m = \overline{S3} = \bar{A} + B$$
$$b = S3 = A\bar{B}.$$

Before implementing the circuit equations, the designer is strongly advised to fill in all blank entries in the state table, as under no condition should a circuit response be left unspecified. The most straightforward method of achieving this is by direct reference to the circuit equations, as we illustrate below.

Our four blank squares in Figure 1.15(b) are collectively defined by Boolean product $A \cdot \bar{B}$, that is by $A = 1$ and $B = 0$.

Substituting the above values in the flip-flop equations, we obtain

$$J_A = B \cdot 5 = 0$$
$$K_A = B \cdot \bar{6} = 0$$
$$J_B = \bar{A} \cdot 4 = 0$$
$$K_B = \bar{4} \cdot \bar{5} + A \cdot \bar{4} = \bar{4} \cdot \bar{5} + \bar{4} = \bar{4}.$$

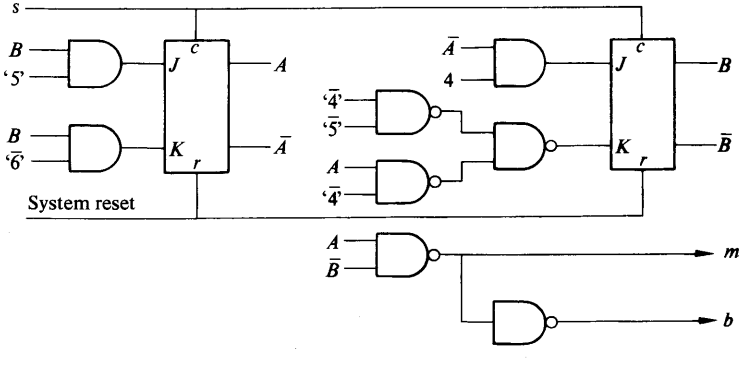


Figure 1.16.

This indicates that should our circuit fail to turn the motor off in state $S3$, JKFFA locks into its set state ($J_A = K_A = 0$) and JKFFB locks into its reset state, since $J_B = 0$. Therefore, we enter $S3$ in the blank squares, as shown in Figure 1.15(a). The corresponding circuit is shown in Figure 1.16.

1.11 REFERENCES

1. Zissos, D. 'Problems and Solutions in Logic Design', Oxford University Press, 1976.
2. Zissos, D. 'Logic Design Algorithms', Oxford University Press, 1972.
3. Zissos, D. 'Race-hazards.' *In* 'Process Control by Power Fluidics', Proceedings of an International Symposium of the Institute of Measurement. Sheffield, U.K., September 1975.
4. Caldwell, S. H. 'Switching Circuits and Logical Design,' Wiley, 1965.
5. Duncan, F. G. and Zissos, D. 'Gate Tolerance in Sequential Circuits', *Proc. I.E.E.*, **118**, No. 2, February 1971.
6. Duncan, F. G. and Zissos, D. 'Design of a Synchronous Multi-level Sequential Circuit', *Proc. I.E.E.*, **119**, No. 2, February 1972.

2

The Microprocessor

In this chapter we discuss in general terms the microprocessor chip, paying particular attention to those characteristics which feature in the design of microprocessor systems. We also outline our design philosophy, which is adopted throughout the book.

2.1 THE MICROPROCESSOR

Definition. A microprocessor is a program-driven clocked sequential circuit,† whose main functions are

(i) to execute programs‡, and
(ii) to control the activities of bus-organized systems (see Figure 2.1)

The operation and function of the system components in our diagram are described later in this chapter.

The program is typically, though not necessarily, stored in semiconductor stores, such as RAMS, ROMS and PROMS. These are discussed later.

Before we have a closer look at microprocessors, it should be pointed out that their internal organization contains no special circuit or architectural features which do not exist in conventional digital computers. For example, an m.p.u. chip will contain registers, arithmetic logic units, decoders, condition flags, and so on. The main difference is that the rapid development in recent years of MOS technology has allowed *more and more circuits to be accommodated in less and less space.* This has created a major problem in l.s.i.

† Clocked sequential circuits are multistate logic circuits whose operation is synchronized with the application of clock pulses, between which no circuit transitions take place.
‡ A program is a sequence of valid instructions used by the microprocessor to execute a given task.

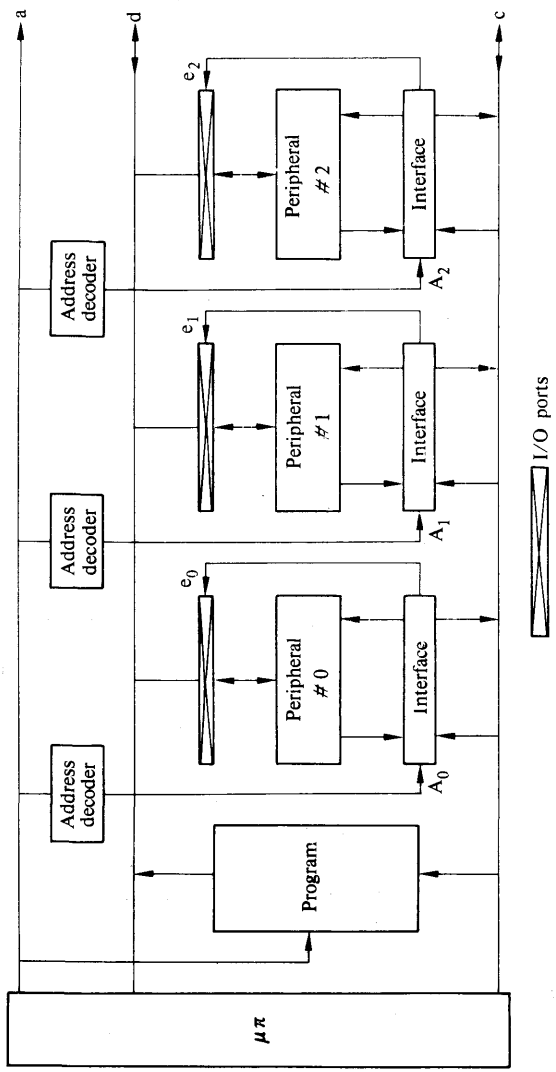

a
d
c
μπ
Program
Address decoder
Address decoder
Address decoder
Peripheral # 0
Peripheral # 1
Peripheral # 2
Interface
Interface
Interface
e_0
e_1
e_2
A_0
A_1
A_2
I/O ports

Figure 2.1.

(large-scale-integration) circuit design, which is that whereas the capacity of the i.c. (integrated circuit) chips for logic components is very large, the number of pins that can be accommodated mechanically on a chip is limited by its physical size (14, 16, 24 and 40 pins on a standard chip). In the case of microprocessors, this problem is overcome by *time-sharing* the input/output pins. For example, the same pins are used to enter data and instructions into a microprocessor, as well as to output data from it. The mechanism by which m.p.u. pins are time-shared in practice will be explained next.

Time Sharing Mechanism

For the sake of clarity we shall omit in this section all details which are not relevant to the understanding of the time-sharing mechanism. Within this context the block diagram of our microprocessor is shown in Figure 2.3 (p. 36). The four registers shown in this diagram are

1. The program counter—*PC*
2. The accumulator—*AC*
3. An addressing register—*r*
4. The instruction register—*IR*

They perform the following functions

PC This is a register which contains the address of the next instruction to be executed.
AC From our point of view this is the register concerned with data transfers in and out of the microprocessor.
r This is an addressing register, that is a register whose contents can be used as an address in a fetch or a store operation.
IR This is the register which receives each instruction in turn, and holds it during execution.

Note that the above registers represent the minimum number of registers that a microprocessor can have. In practice, microprocessors have additional registers, such as stack pointers, index registers, etc. The function of these registers and the internal organization of microprocessors is discussed in 'Digital Interface Design' (2nd edition) [1].

Our microprocessor executes an I/O instruction in three machine cycles, *M*1, *M*2 and *M*3, as shown in Figure 2.2. (Machine cycle 2 may be executed

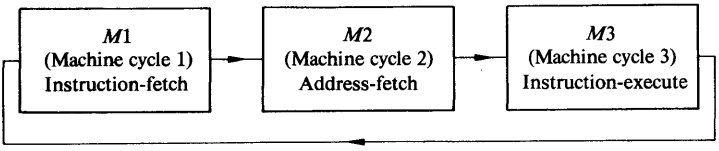


Figure 2.2.

twice if the byte is, say, 8 bits and the address 16 bits). In our case, machine cycle $M1$ has four internal states, and machine cycles $M2$ and $M3$ have three states, as shown in Figure 2.4.

If we assume that the program counter (*PC*) has been loaded with the address of the next instruction, the cycle of operations comprising the execution of an I/O instruction is as follows. (During this discussion the reader will find it helpful to make constant reference to Figures 2.3 and 2.4).

Machine cycle 1

During this machine cycle, also referred to as the *instruction fetch cycle*, the next instruction to be executed is transferred from memory (ROM or RAM) into the instruction register, in the following manner.

If we denote by A_m the location in memory in which the next instruction resides, the contents of the program counter (*PC*) prior to the execution of this cycle will be A_m, as stated on page 34. Now, when our microprocessor enters state $M1 \cdot T1$ in Figure 2.4 the address bus is connected through tristates within the m.p.u. chip to the *PC*, allowing the address signals defining A_m to be output. The appropriate sequence of electronic signals are next generated, which cause the contents of location A_m (our next instruction) to be released, and appear on the output terminals of the memory chips. During this machine state the data bus, *d*, carries no information and is therefore tristated.

On the next clock pulse our microprocessor moves to state $M1 \cdot T2$. Nothing changes in this state; that is the program counter (*PC*) remains connected to the *a* bus and the *d* bus remains tristated. The function of this state, as we shall see in the next section, is to provide the user with the opportunity of delaying entry of the microprocessor into state $M1 \cdot T3$ in Figure 2.4, before the memory, or whatever medium contains the next instruction, has had time to respond. As the problem of memory and I/O synchronization is discussed in detail in the next section, we shall assume at this stage that both the medium containing the program and the peripheral respond fast enough not to require slowing down of the microprocessor.

When the microprocessor enters state $M1 \cdot T3$ the data bus is connected to the instruction register (*IR*) through a set of tristates within the m.p.u. chip. Simultaneously, we connect the output terminals of the memory that contains the program to the data bus, thus establishing a direct link between the memory and the instruction register, as shown in Figure 2.3. This means that the next instruction is now available at the input of the instruction register (*IR*). A pulse is then generated within the m.p.u. which loads this instruction into the instruction register.

The next clock pulse moves the microprocessor to state $M1 \cdot T4$ in Figure 2.4. In this state the contents of the instruction register (that is the instruction) is

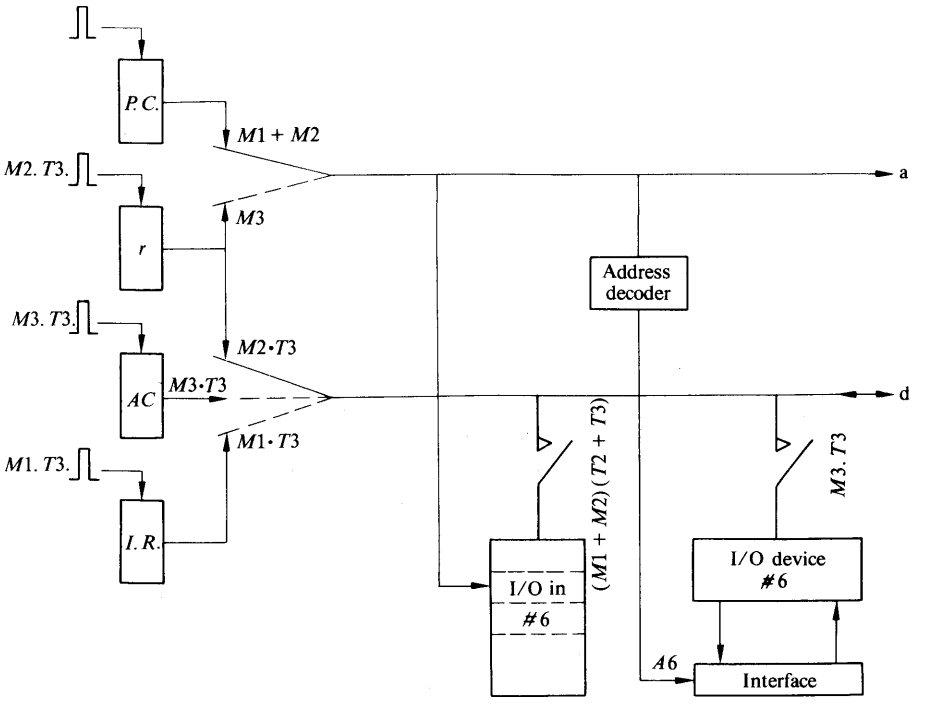


Figure 2.3.

decoded, the program counter (*PC*) is incremented and machine cycle *M*2 is entered on the fourth clock pulse, as shown in Figure 2.4.

Machine cycle 2

This cycle is also referred to as *address fetch cycle*, because this is the cycle used to transfer the address of the I/O device from memory into the m.p.u. The mechanics of transfer are identical to those used to move the instructions from memory into the m.p.u. during the previous cycle. Throughout this cycle the program counter (*PC*) is connected to the *a* bus.

In state $M2 \cdot T1$ the signals on the *a* bus cause the contents of the next highest location in memory to be released. As in $M1 \cdot T1$ the *d*-bus is tristated.

In state $M2 \cdot T2$, as in the case with state $M1 \cdot T2$, nothing changes. This state, as we have already explained, is used for synchronization purposes discussed in the next section.

When our microprocessor enters state $M2 \cdot T3$, the data bus is connected to

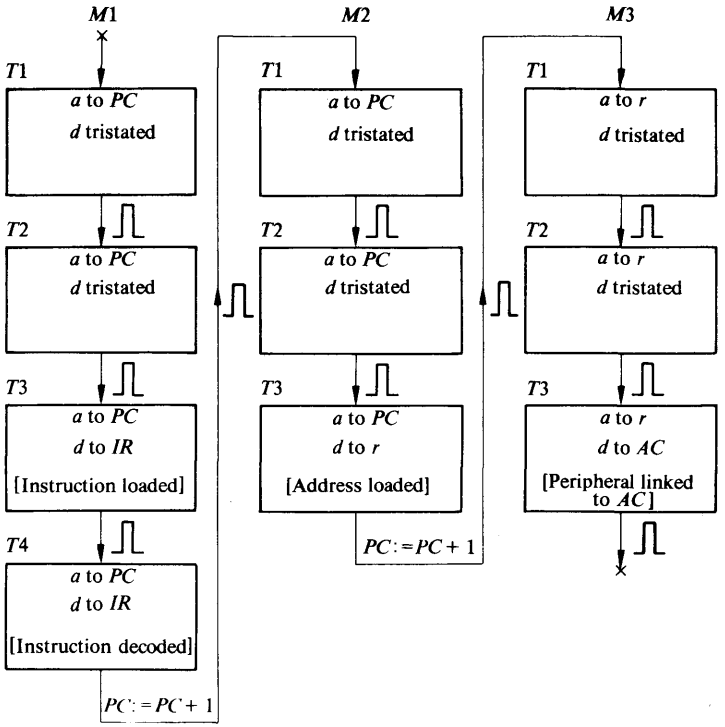


Figure 2.4.

the addressing register (r) through a set of tristates within the m.p.u. chip. Simultaneously, we connect the output terminals of the memory that contains the program to the data bus, thus establishing a direct link between the memory and the addressing register (r), as shown in Figure 2.3. This means that the I/O address is now available at the input of the addressing register. A pulse is next generated within the m.p.u. which loads the I/O address into the addressing register, the program counter (PC) is incremented and machine cycle $M3$ is entered on the seventh clock pulse, as shown in Figure 2.4.

Machine cycle 3

This cycle is also referred to as *I/O execute cycle*, because during this cycle a direct link is established between the microprocessor and the peripheral, specified by the I/O address.

During machine cycle 3 the address bus is connected to the addressing register, r, allowing the I/O number to be present on the address bus for the whole duration of this cycle.

In states $M3 \cdot T1$ and $M3 \cdot T2$, as in the case of the corresponding states in the previous two cycles, the data bus is tristated. In state $M3 \cdot T1$ the I/O address is made available to the system and in state $M3 \cdot T2$ the system designer is provided with the opportunity to delay the entry of the microprocessor into state $M3 \cdot T3$ in Figure 2.4. This may be needed for synchronization purposes, since, as we have already explained, the m.p.u. must not enter state $M3 \cdot T3$ until the peripheral is ready. The mechanics for implementing such a delay are explained in the next section. For the sake of simplicity at this stage we shall assume that no synchronization is needed.

In state $M3 \cdot T3$, our tenth and last state, the data bus is connected to the Accumulator (AC) through a set of tristates within the m.p.u. chip. At the same time, we connect the data terminals of our peripheral to the data bus, thus establishing a direct data link between it and the Accumulator, as shown in Figure 2.3. During this time data is transferred in either direction, depending on whether an input or an output instruction is being executed.

2.2 WAIT STATES

Memory Synchronization

As already explained, the microprocessor is a clocked sequential circuit, whose processing activities are timed by a clock, as shown in Figure 2.4. Clearly the higher the clock frequency, the faster the system. The maximum clock frequency that can be used in a given system is determined by the response time of the internal circuits of the m.p.u. and by the access time of the memory chips storing the programs. With present-day components the limiting factor in practice is usually the memory access time. It is therefore essential that the system designer understands the mechanics of memory synchronization, if he wishes to work at the maximum allowable clock frequency. A step-by-step explanation is given below.

During an instruction or an address fetch cycle, that is during a cycle in which the memory must supply an addressed byte to the m.p.u., the microprocessor clock frequency must clearly be low enough to allow the memory to respond before it starts its next activity. In the case of our microprocessor, whose I/O cycle is shown in Figure 2.4, the m.p.u. sends a read address to the memory in state $M1 \cdot T1$. In state $M1 \cdot T3$, that is two clock periods later, the output of the selected memory chip is read into the instruction register (IR), as shown in Figure 2.3. It is obvious that we must not allow the m.p.u. to enter state $M1 \cdot T3$ before the addressed memory chip has had time to respond. Therefore, in our case the maximum clock frequency, f_{max} must be so chosen that two clock periods are always greater or equal to the access time of the memory chips, denoted by variable t.

Expressing the above condition algebraically, we obtain

$$\frac{2}{f_{\max}} \geq t$$

or

$$f_{\max} \leq \frac{2}{t}$$

Note that memory synchronization is not a problem that the system designer usually faces, because in practice the m.p.u., the clock and the memory chips are selected from the same family of components, with the maximum clock frequency specified by the manufacturer. If the designer wishes, for some reason or other, to implement his own basic system using components not belonging to the same family, the methods used to interface peripheral devices can be employed.

I/O Synchronization

As in memory cycles the microprocessor operation must be synchronized with the access time of the memory, so in an I/O execute cycle (see Figure 2.3) the microprocessor must be synchronized with the response time of the peripheral in question. For example, if during an I/O operation we have to advance a paper tape and read the next character, we must clearly prevent the microprocessor from executing a read operation until the tape has been advanced to its next character position. In terms of our microprocessor, this means that we must prevent it from entering state $M3{\cdot}T3$ in Figure 2.4 until the new character is ready to be read. We can achieve this by activating the reader when the microprocessor enters state $M3{\cdot}T2$, that is immediately after the reader's address is output on the address bus, and then prevent it from moving to its next state, $M3{\cdot}T3$, until the new character is available. As we shall explain in Chapter 3, the most straightforward method of achieving this would be to turn off the microprocessor clock during the time that the peripheral in question is responding. With the exception of a small number of microprocessors, such as the Signetics 2650 and the RCA 1800 series, it is not possible to turn off the clock of present-day microprocessors without destroying the electrical state of the m.p.u. To overcome this constraint, microprocessors are provided with a synchronization feature which allows them to enter a *wait state* in which the microprocessor idles without turning off the clock. A wait period may be of indefinite duration.

Returning to our example of advancing and reading a paper tape, we can synchronize the microprocessor operation with the reader response using the wait state in the following manner.

When state $M3{\cdot}T2$ in Figure 2.4 is assumed we prevent the microprocessor from moving to state $M3{\cdot}T3$ and direct it instead to the wait state, as shown in Figure 2.5 (a). When the microprocessor enters the wait state we activate the

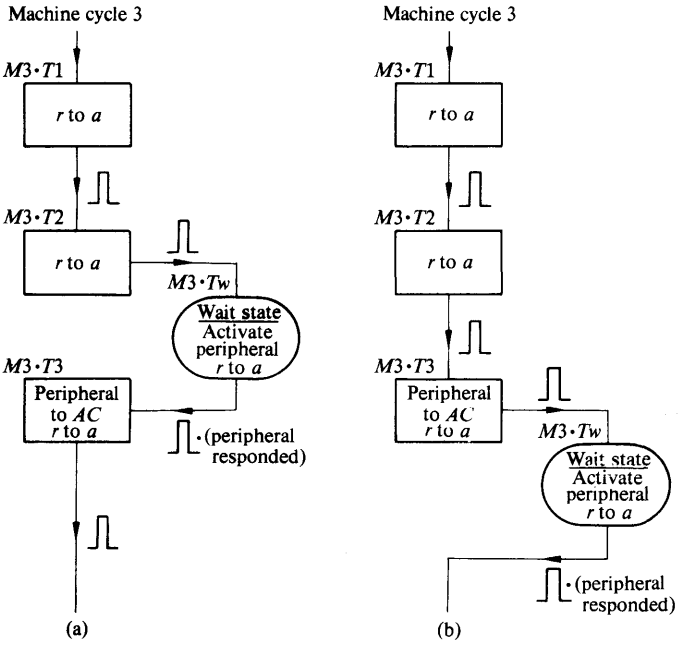


Figure 2.5.

peripheral, in our case a reader. When the peripheral responds, its ready signal changes to 0 and remains at this level while the peripheral is responding. It changes back to 1 when the peripheral has fully responded. We use the 0 to 1 transition of signal r to move the microprocessor out of the wait state into state $M3 \cdot T3$ in Figure 2.5 (a). This causes the microprocessor to assume its normal operation, that is the tape is read in state $M3 \cdot T3$, after which it starts to execute its next cycle.

If we modify our problem to one of read and advance the tape, we would clearly wish to enter a wait state after the tape is read, that is after we leave state $M3 \cdot T3$, as shown in Figure 2.5 (b).

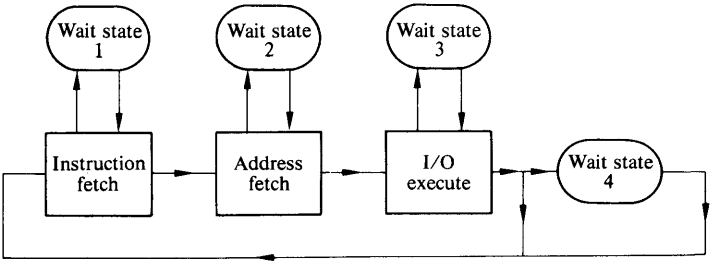


Figure 2.6.

As before, when the microprocessor enters the wait state we activate the peripheral. We keep it in the wait state until the peripheral has fully responded which is indicated by its ready signal r changing from 0 to 1. On leaving the wait state the microprocessor moves to state $M1.T1$ in Figure 2.4, which allows it to resume its normal operation.

Reference to our paper tape reader shows that two types of I/O synchronization are needed. One in which the peripheral is first activated and then accessed by the microprocessor (*advance and read* in our example) and another in which the peripheral is first accessed and then activated (*read and advance* in our example). We shall refer to these two types as

(i) activate and access, and
(ii) access and activate.

In Figure 2.6 we show the block diagram of an I/O cycle of a microprocessor with provisions for entering a wait state during any machine cycle or at the end of an I/O instruction. This allows for memory as well as for both types of I/O synchronization.

Not all microprocessors have the wait state configuration shown in Figure 2.6. For example the INTEL 8080 does not have wait state 4, while the Motorola 6800 has only wait state 4[2, 3]. See Figure 2.7.

Entry and exit from wait states is controlled usually, but not exclusively, by applying prescribed logic levels to specified pins on the m.p.u. chip. For

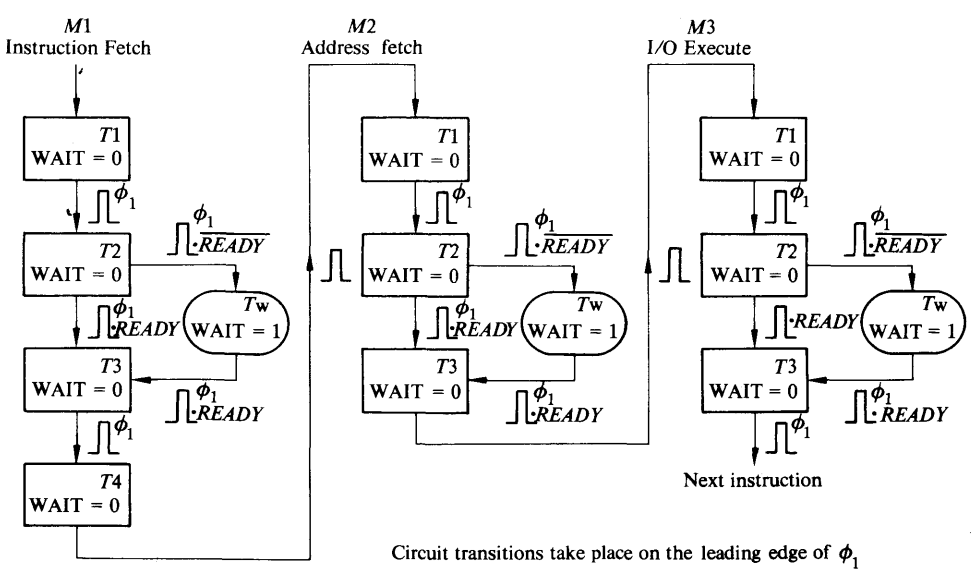


Figure 2.7.

example in the case of the INTEL 8080 a logic '0' on pin 23 (its READY line) causes it to enter the wait state, while a logic '1' on the same pin moves it out of the wait state—see Figures 2.7 and 2.9. In the case of the Motorola 6800 it enters the wait state either by applying a logic '0' on pin 2 (its HALT line) or by executing a 'wait for an interrupt' 3E instruction. It moves out of the wait state either by applying a logic '1' to pin 2 or by generating an interrupt request—see Figures 2.8 and 2.10.

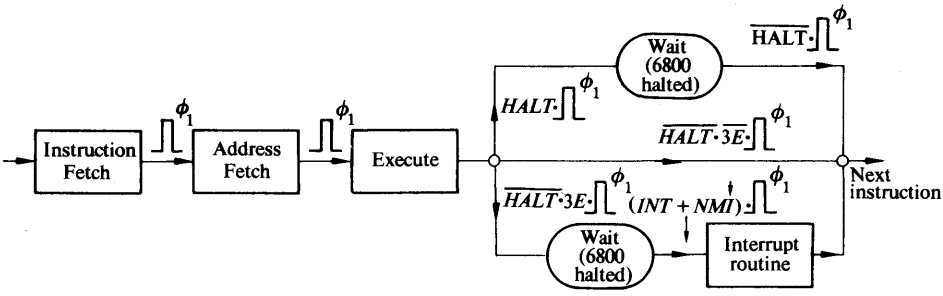


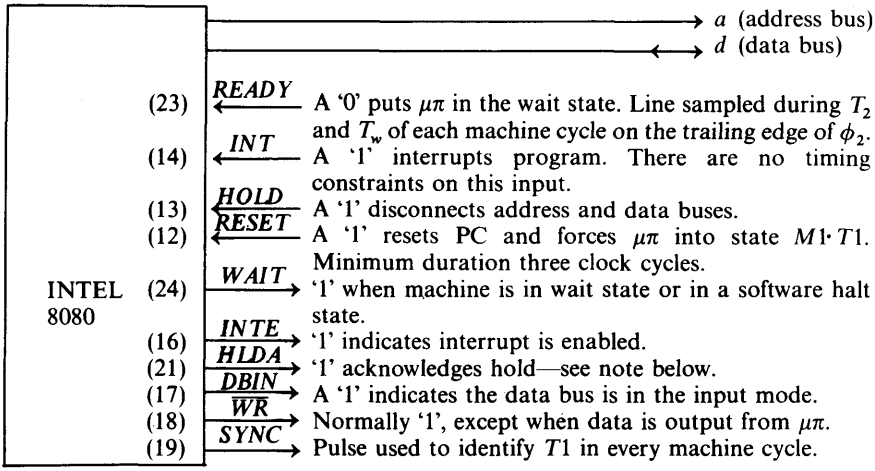
Figure 2.8.

2.3 M.P.U. SIGNALS

Like all sequential circuits the microprocessor has internal states and it responds to external signals. The external signals to which it responds, as in the case of all other circuits, are referred to as *command signals*. To allow the system designer to synchronize the operation of peripherals to that of the microprocessor, the m.p.u. generates *status signals*. These signals do not give a direct indication of the internal state of the machine, as would be the case if state variables (secondary signals) were used.

The designer must ensure that he has interpreted correctly the status signals of a microprocessor, before using it. Although this applies to all equipment one uses in a system, it is particularly essential in the case of microprocessors. This is because, as we have explained in the previous section, the data and address pins in a microprocessor chip are time-shared by several components of the m.p.u. in an instruction cycle, making the timing of the command signals in relation to the status signals critical.

The command and status signals of a microprocessor are collectively referred to as *m.p.u. signals*. The m.p.u. signals vary widely from microprocessor to microprocessor. We illustrate this by showing in Figures 2.9 and 2.10 the m.p.u. charts of the INTEL 8080[2] and the MOTOROLA 6800[3].



NOTE. This signal goes high within a specified delay of the leading edge of $\phi 1$. The address and data buses are floated within a brief delay after the rising edge of the next $\phi 2$ clock pulse.

STATUS WORD CHART

Type of machine cycle

data bus bit	status information	(1) instruction fetch	(2) memory read	(3) memory write	(4) stack read	(5) stack write	(6) input read	(7) output write	(8) interrupt acknowledge	(9) halt acknowledge	(10) int. acknowledge while halt
											←N STATUS WORD
D_0	*INTA*	0	0	0	0	0	0	0	1	0	1
D_1	$\overline{WO}$	1	1	0	1	0	1	0	1	1	1
D_2	*STACK*	0	0	0	1	1	0	0	0	0	0
D_3	*HLTA*	0	0	0	0	0	0	0	0	1	1
D_4	*OUT*	0	0	0	0	0	0	1	0	0	0
D_5	M_1	1	0	0	0	0	0	0	1	0	1
D_6	*INP*	0	0	0	0	0	1	0	0	0	0
D_7	*MEMR*	1	1	0	1	0	0	0	0	1	0

Figure 2.9

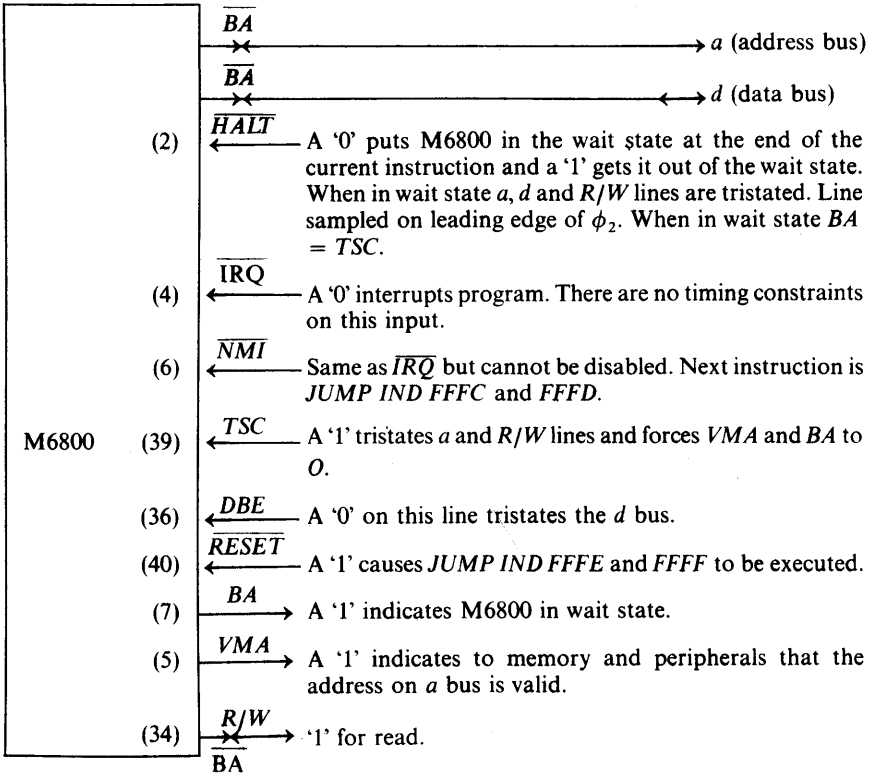


Figure 2.10.

2.4 MODES OF MICROPROCESSOR OPERATION

From the system designer's point of view the microprocessor can operate in any of the modes listed below.

1. The Internal Mode
2. The Wait/Go Mode
3. The Test-and-Skip Mode
4. The Interrupt Mode
5. The D.M.A. Mode
6. The D.D.T. Mode.

With the exception of the internal mode, all the modes listed above are discussed in detail in the following chapters. In this section we give only a brief description of their characteristics.

The Internal Mode

In this mode the instructions and data reside in the system's ROMS and RAMS. As no peripherals are involved, we shall not discuss this mode of operation any further. For a detailed description of this mode the interested reader is referred to[1].

The Wait/Go Mode

In this mode the internal operation of a microprocessor is synchronized with the slower response of devices, by the microprocessor entering *a wait (idle) state* while the peripheral being accessed is responding (see Figure 3.3). The mechanics of putting the microprocessor in a wait state have been explained in the previous section. As we shall see in Chapter 3, the main features of this mode are

1. Timing problems are eliminated.
2. Design time and effort are minimal compared to other modes.
3. The interface hardware is minimal.†
4. The 'wait' and 'go' are everyday concepts that we all understand.
5. Maintenance is easy.

The disadvantage of this mode, in common with the 'test-and-skip' mode, is that the microprocessor idles during the time that the peripheral being accessed is responding. In some applications this may be either undesirable and/or intolerable.

The Test-and-Skip Mode

Functionally, this mode is the same as the wait/go mode with the following exception. Synchronization of the microprocessor with a peripheral is implemented by causing the microprocessor to enter a *software loop* during the time that the peripheral being accessed is responding. In this loop the microprocessor inputs and tests the status of the peripheral. If the peripheral is found to be unready the process is repeated, otherwise normal execution of the program is resumed (see Figure 4.1).

The Interrupt Mode

This mode is used to interrupt the execution of the current program in a microprocessor by means of an external signal, *the interrupt signal*, and

† It consists of two wires for action/status devices.

execute a different set of instructions, *the interrupt routine*, requested by an external source. At the end of the interrupt routine the interrupted program is resumed at the point of interruption, as shown in Figure 5.1.

As we shall see in Chapter 5, the design and implementation of this mode, although carried out in well-defined steps, requires relatively more complicated hardware and software than any other mode. Its main advantage is fast system throughput.

The D.M.A. Mode

D.m.a. is the abbreviated form of *direct memory access*. In this mode a direct link is established by the programmer between a peripheral and the memory of the system, as shown in Figure 6.1 (b), whenever we wish to transfer data between them. This mode is particularly suitable when we wish to transfer large blocks of data between a peripheral and memory. Although initiated by the programmer, the transfer of data takes place autonomously, that is without programmer intervention. Usually, though not necessarily, at the end of the block transfer, the interface generates an *end-of-block-transfer* flag, ε, to inform the programmer that the specified block of data has been transferred.

Contrary to common belief, the design and implementation of d.m.a. interfaces is straightforward, as we shall see in Chapter 6. The interface hardware is uncomplicated and the software required to drive it minimal—approximately a dozen instructions for each block transfer.

The reader's attention is also drawn to the fact that most of us have been conditioned to associate the d.m.a. mode with fast mass storage devices and to exclude slow devices, such as paper tape readers, readers, tape punches and so on. As we shall prove in Chapter 6, the d.m.a. mode can be used for block transfers of data between memory and any type of peripheral, irrespective of how slow it is.

The D.D.T. Mode

D.d.t. is the abbreviated form of *device-to-device transfers*. This is analogous to the d.m.a. mode, insofar that a direct link is established between two or more peripherals in a microprocessor system that allows them to communicate with each other directly (see Figure 7.3).

This mode would be used, for example, in a situation where one might wish to obtain a hard copy of some data stored, say, on a tape. Because each transfer of a byte of information in or out of a microprocessor requires a number of instructions to be executed, it is clearly wasteful both of time and instructions to move data in and out of a microprocessor if no processing of the data is

required. In situations like this a direct link be established between peripherals.

The formalization of the implementation of this mode is based on the use of sequential equations. These equations are discussed in Chapter 1.

2.5 SEMICONDUCTOR MEMORIES

Semiconductor memories are available in the form of integrated circuits in standard dual-in-line packages. Each such package, a chip, is described as containing a number of *bits*, almost always a power of 2. These bits may be organized in groups (bytes†) of 8 (2^3) or some other small power of 2. Memory chips are described as $2^m \times 2^n$; this implies that 2^{m+n} bits can be stored and that they are organized in 2^m bytes and that each byte has 2^n bits. In such a case there will be *m address lines* to allow each byte to be individually addressed. Thus a memory chip described as 128×8 ($2^7 \times 2^3$) implies it contains 1024 (2^{10}) bits and that the bits are organized in 128 eight-bit bytes. In this case there will be seven address lines by means of which any individual byte can be selected. Similarly 4096×1 describes a memory chip of 4096 bits, with 12 address lines, enabling any individual bit to be separately addressed.

In addition to address terminals, the memory chips are usually provided with *chip select terminals*. These terminals are used to identify one or more chips that constitute a memory module in a system.

How the microprocessor address signals are split between chip select and address lines depends on the system architecture.

Semiconductor memories are classified as ROMS, EPROMS, PROMS and RAMS. A brief description of each type is given below.

ROMS

This is the abbreviated form of *read-only-memories*. These are memory chips that contain information which has been built into it during manufacture. Such information is permanent and it cannot be erased and replaced by new information. Therefore the information which is stored in ROMS is limited to specialist uses, such as storing standard programs, code-conversion (look-up) tables, etc. Unlike RAMS, ROMS retain their information when power is switched off.

The main advantages of ROMS are large bit-capacity, low power, fast access time and their non-volatile nature, while their disadvantages are their limited

† A byte is a number of bits treated as an entity.

use and the fact that, because the information cannot be changed, a single error can be costly.

The block diagram of a ROM is shown in Figure 2.11. The release of the addressed byte onto the microprocessor data bus, or onto the tristate output terminals, is timed by the enabling signal of the tristates. Clearly this signal must be applied after the memory chip has had time to respond to the address signals. In practice one of the chip select signals is used for this purpose. We shall refer to this signal as an *action signal.*

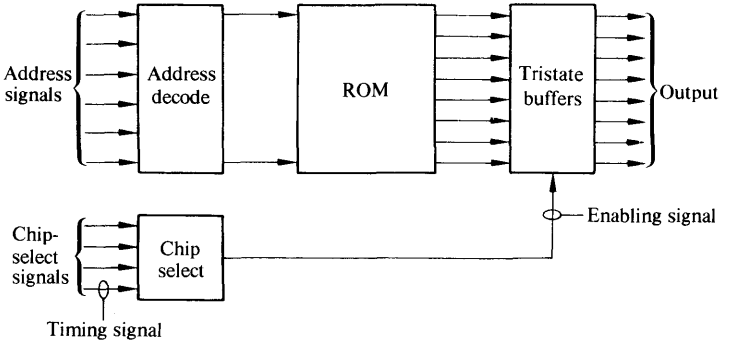


Figure 2.11.

EPROMS

This is the abbreviated form of *erasable-programmable-read-only-memories.* When it is installed in a system, this device behaves exactly like a ROM. The significant difference between a ROM and an EPROM is that the latter can be removed from the system and 'reprogrammed'. This means that the information in it can be erased and replaced by new information. The reprogramming process requires specially-designed apparatus, typically an ultra-violet radiation source for erasure and a source of high voltage pulses for re-writing. After the initial information has been erased, the new information is written into the chip by subjecting each single memory cell to a very high voltage. This causes the corresponding capacitor to charge and remain charged for approximately 100 years.

EPROMS in practice are used for storing semi-permanent information or permanent information originated by the user himself.

PROMS

Programmable read only memories. When installed in a system, PROMS

like EPROMS, behave like ROMS. To store a '1' in a selected bit cell, a high current is used to blow the corresponding fuse, thus creating a permanent open circuit. They can only be used once.

They are not as reliable as EPROMS, particularly for process control and industrial applications.

RAMS

Random-access-memories. Information can be both *written* and *read* from it under program control. The block diagram of a RAM is shown in Figure 2.12. As in the case of ROMS it contains address pins for accessing a particular location within the RAM, and chip select signals for identifying the chip in a system (or the memory module to which it belongs).

Two control signals are normally required by a RAM

(i) a *read/write* (*R/W*) *signal* to define the next operation, and
(ii) as in the case of ROMS, an action signal to control the timing of the transfer.

In practice the control signals provided by different manufacturers vary.

The information stored in RAMS is volatile, that is, it is lost when the power is turned off. Some RAMS, however, can be operated in a 'stand-by' mode, defined by another control signal. In this mode the RAM is disabled but it uses reduced power while retaining its information.

RAMS can be either *static* or *dynamic*. In a static RAM, as long as the power is maintained, information once written is held indefinitely without special provision by the designer. In a dynamic RAM, however, the information written into it will be lost within a matter of a few milliseconds unless it is

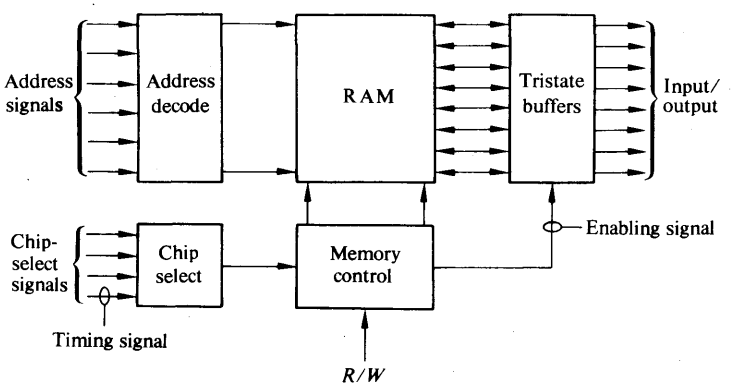


Figure 2.12.

refreshed. This is because a capacitive effect is used as the means of storage, and the charges leak. Therefore they have to be 'topped up' before a '1' is indistinguishable from '0'.

Static RAMS have lower capacity and are slower than dynamic RAMS, but they do not need refreshing circuitry. Dynamic RAMS, generally speaking, are found to be not as reliable as static RAMS.

STACKS

A stack is a block of consecutive addresses in RAM which can be accessed from one end on a *last-in-first-out* (lifo) basis. In microprocessor systems the stack address is generated by the *stack pointer*. This is an up/down counter, which is normally stepped down after each loading (push) operation and stepped up after each retrieve (pop) operation.

In a given system a block of consecutive memory locations is dedicated to stacking operations. If not done automatically, the user must initialize the stack pointer to the first stack location.

2.6 I/O PORTS

A block diagram showing the tristate arrangements implementing input and output ports is shown in Figure 2.13 (a). When enabling signal $e_1 = 1$ the terminals of the input port are connected to the address bus, allowing a source to write data onto it. Similarly when $e_2 = 1$ the terminals of the output port are connected to it, allowing an acceptor to read data from the bus. (When an enabling signal e equals 0 the tristates are open circuited). Clearly while digital information travels from one source to one or more acceptors, all other sources that are tied to the bus must be disabled.

The tristate arrangement connecting to the bus a device that can act both as a source and as an acceptor is shown in Figure 2.13 (b). When $e_1 = 1$ and $e_2 = 0$ the device can read data from the data bus, when $e_1 = 0$ and $e_2 = 1$ it can write data on the bus, and when $e_1 = e_2 = 0$ it is disconnected from the bus. The use of I/O ports in microprocessor systems is shown in Figure 2.1.

Note that our device has one set of lines only, which are used both for reading and writing. These lines are called *bidirectional lines.*

2.7 ADDRESS DECODERS

In our block diagram in Figure 2.1 for the sake of clarity we have used local address decoders. These are essentially AND gates that produce an output

when signals of prescribed levels are applied at their input. For example a decoder for address 6, assuming a four-bit address, consists of two inverters and an AND gate, as shown in Figure 2.14 (a).

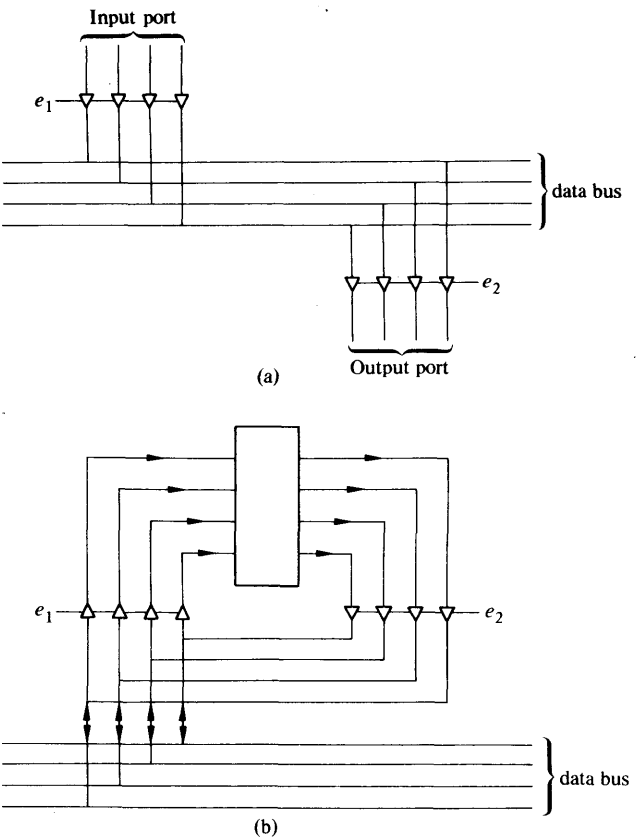


Figure 2.13.

In practice i.c. chips are available for address decoding. A commonly-used i.c. chip for this purpose is one that decodes three address lines to eight addresses—see Figure 2.14 (b). A larger number of address lines can be decoded by suitably interconnecting i.c. chips, using the methods described in Chapter 1.

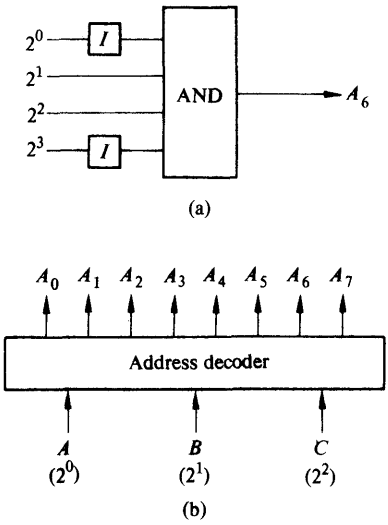


Figure 2.14.

2.8 INTERFACES

The function of an interface is to monitor the state of two or more devices between which data is to be transferred and to issue the command signals for each device in the correct sequence. The sequence is usually but not necessarily programmed. In Figure 2.15 we show the block diagram of an interface between a source and an acceptor.

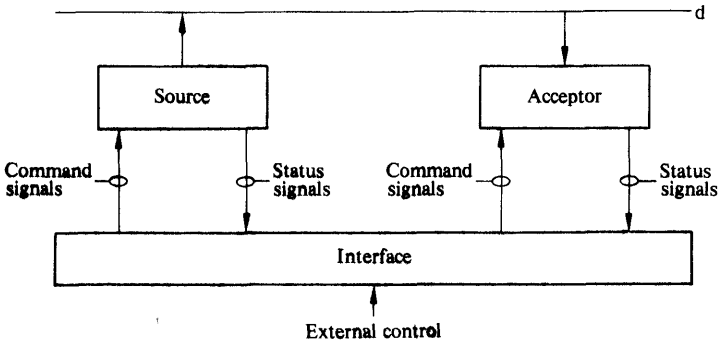


Figure 2.15.

Formally an interface is defined as *the set of circuits, signals and procedures required to effect a transfer of data between digital devices.*

2.9 DESIGN PHILOSOPHY

The design philosophy adopted is one that allows the inexperienced user to produce sound and reliable systems simply, while at the same time providing the specialist with the tools to improve his technique in dealing with more sophisticated assemblies. As in the case of logic circuits elegance of design is not sought, but can be achieved.

In developing our design philosophy, we considered the following as important.

1. *System reliability.* All systems must function correctly.
2. *Circuit maintainability.* The systems should be easy to maintain.
3. *Design effort.* This must be minimal to allow for greater creativity.
4. *Documentation.* This should be concise and to the point. Symbols and diagrams are preferable to verbal statements; they are more readily understood by non-English speaking people and are likely to prove more attractive to the export market.
5. *Design steps.* These must be easy to apply. In our case no specialist knowledge is necessary.
6. *Modifications.* The systems should be easily modifiable to meet new conditions as they arise.

2.10 DESIGN STEPS[1]

The design process is accomplished in five steps, listed below (see also Figure 2.16).

Step 1 *Aim of the design*

The system specification is expressed in the designer's terms. This step is introduced to ensure that the system requirements are interpreted correctly by the system designer.

This stage is critical for successful cooperation between the system designer and the user. Failure at this stage is usually the cause of system misoperation which then produces the need for subsequent design modifications.

Step 2 *Device characteristics*

In this step the designer studies the terminal characteristics of the devices to be used. Any consideration of purely internal characteristics should be avoided.

Step 3 *System design*
In step 3 the designer specifies the system characteristics in general terms by means of a block diagram and a system flow chart.

Step 4 *Hardware design*
The fourth step is provisional, and its results may well be modified in the light of the experience of the next step. It is accomplished conventionally, using well-established methods.[4]

Step 5 *Software design*[5]
On the basis of the hardware design in step 4 and assuming the necessary machine code instructions, the basic software for the operation of the device is designed. This process may well indicate improvements to the hardware which was designed in step 4. In fact, steps 4 and 5 should be regarded as complementary, and should be repeated until a satisfactory design is obtained.

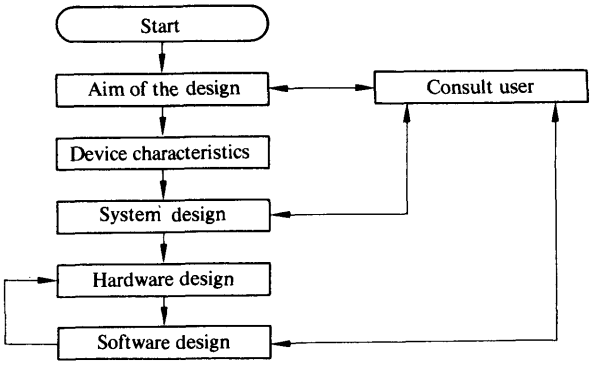


Figure 2.16.

2.11 REFERENCES

1. Zissos, D. and Duncan, F. G. 'Digital Interface Design', (2nd edition) Oxford University Press, (to be published).
2. INTEL 8080 Microprocessor User's Manual, September 1975.
3 M6800 Microprocessor System Design Data, Motorola, 1976.
4. Zissos, D. 'Problems and Solutions in Logic Design', Oxford University Press, 1976.
5. Duncan, F. G. 'Microprocessor Programming and Software Design', Prentice-Hall, 1979.

3

Wait/Go Systems

In this chapter we explain the wait/go concept and use it to design microprocessor systems. The design of wait/go systems requires no specialist knowledge of electronics or of microprocessors and, therefore, can be undertaken by the user with no expertise in these areas. The design philosophy adopted and the design steps are outlined in sections 9 and 10 of Chapter 2.

3.1 INTRODUCTION

As we mentioned in Chapter 2, during an I/O (input/output) operation it is necessary to synchronize the microprocessor cycle with the response of the peripheral. For example, if a microprocessor outputs one byte of information every ten microseconds, but the acceptor, say a printer, takes 100 microseconds to print a byte, clearly nine out of ten bytes will be lost, unless the microprocessor is slowed down. We must therefore ensure that in any design the microprocessor does not attempt to drive a peripheral faster than it can go.

Synchronization between a microprocessor and a peripheral under these circumstances is traditionally achieved, as with minicomputers, by the microprocessor entering a software loop while the device is responding. This method, which is explained in the next chapter, is referred to as *test-and-skip.*

In a limited number of cases, where I/O synchronization can be achieved by slowing down the microprocessor clock frequency, a method known as *clock-stretching*[1] can be used. This method is also explained in the next chapter.

More recently a third method of achieving I/O synchronization was developed by Zissos and Duncan,[2]. In this method the internal operation of a microprocessor is synchronized automatically with the response of slower peripherals, thus eliminating the need for synchronization signals. This allows microprocessor systems to be implemented simply and reliably. Furthermore,

because the 'wait' and 'go' are concepts that we all use everyday, such systems can be designed by the user who may not possess any specialist knowledge of electronics or of microprocessors, such as physicists, chemists, mechanical engineers, medical experts and so on.

The main properties of wait/go systems are listed below.

From the system designer's point of view, these are

1. Design time and effort are minimal.
2. Interface hardware is minimal. It consists of two-wires for action/status devices.†
3. Timing problems are automatically eliminated.
4. Its speed is comparable to that of the conventional 'test-and-skip' mode.
5. The software is reduced.

From the user's point of view, the main features are

1. High degree of reliability as a result of minimal hardware.
2. The 'wait' and 'go' are everyday concepts that we all understand.
3. The design of 'wait/go' systems does not require specialist knowledge of 'electronics', thus allowing the average user to specify and design his own systems.
4. High degree of transparency.
5. It can be readily introduced as an 'add-on' feature to existing systems.
6. Maintenance is easy.

3.2 THE WAIT/GO CONCEPT

When an I/O (input/output) instruction is recognized, the microprocessor enters automatically a *wait state*. This, as we saw in Chapter 2, is a microprocessor state in which all m.p.u. activities are suspended without turning off the clock. When the microprocessor enters the wait state, signal *w* on the *wait line* changes from 0 to 1—see Figure 3.1. Exit control from the wait

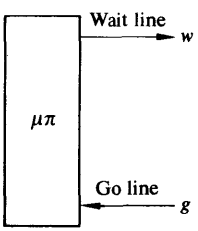


Figure 3.1.

†Action/status devices are discussed in Appendix 1.

state is passed on to the *go line*, *g*. In our case a 0 to 1 signal transition on line *g* takes the microprocessor out of the wait state.

3.3 WAIT/GO SYSTEMS

The block diagram of a wait/go system is shown in Figure 3.2. Its operation is as follows. When an I/O (input/output) instruction is detected the microprocessor enters automatically a wait state and the peripheral is activated. The microprocessor remains in the wait state until the peripheral has fully responded, at which time it assumes its normal cycle, as illustrated in Figure 3.3.

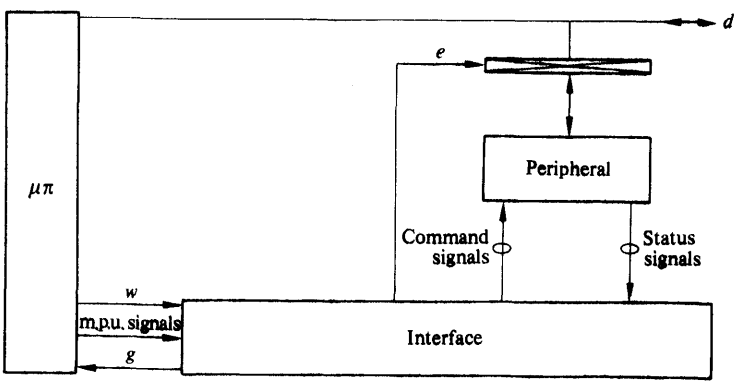


Figure 3.2.

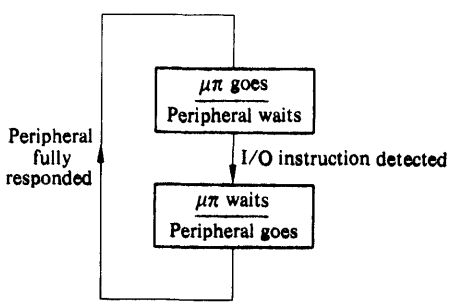


Figure 3.3.

It has been shown that in the case of action/status devices (see Appendix 1), the interface in Figure 3.2 consists of two wires [2]. We shall reproduce the proof below.

Our starting point is the diagram in Figure 3.4. The signals w, g, a and r have the following meaning.

Signal w: A '1' on this terminal (the wait line) indicates that the microprocessor has entered the wait state.

Signal g: A signal transition from 0 to 1 on this terminal (the 'go' line) puts the microprocessor out of the wait state.

Signal a: A signal transition from 0 to 1 on this line triggers the peripheral into action.

Signal r: While the peripheral is responding $r = 0$. When the peripheral has fully responded r changes to 1. No activation is possible when $r = 0$.

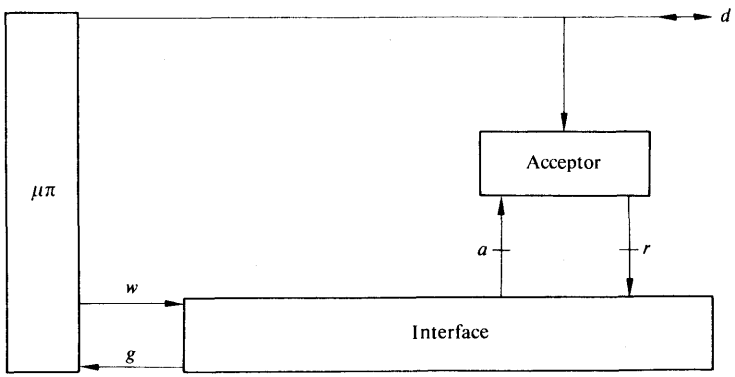


Figure 3.4.

A suitable internal-state diagram of a circuit to implement the above interface is shown in Figure 3.5. Applying the reduction steps† to its equivalent state table in Figure 3.6(a) allows its three rows to merge into one, as shown in Figure 3.6(b).

By direct reference to the reduced state table, we obtain the following equations

$$a = wr + w\bar{r} + (\bar{w}\bar{r}) = w \quad (1)$$

$$g = \bar{w}r + wr + (\bar{w}\bar{r}) = r \quad (2)$$

† The state reduction steps are explained in section 7 of Chapter 1.

The corresponding circuit implementation is shown in Figure 3.7. *That is the interface required between a wait/go microprocessor and an action/status acceptor consists of two wires.*

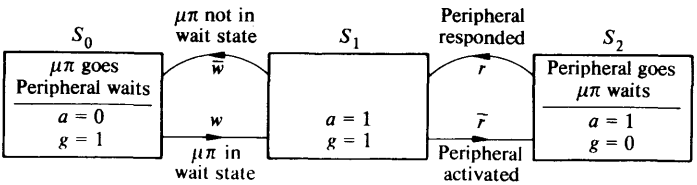


Figure 3.5.

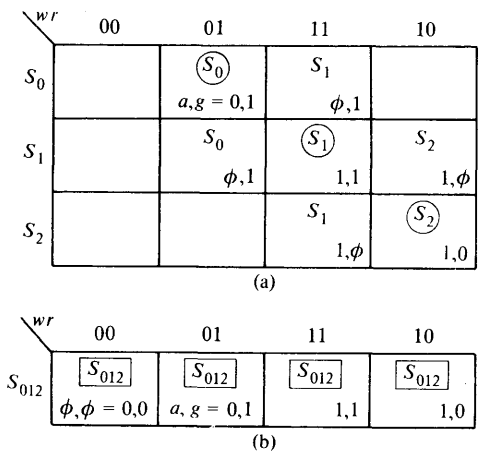


Figure 3.6.

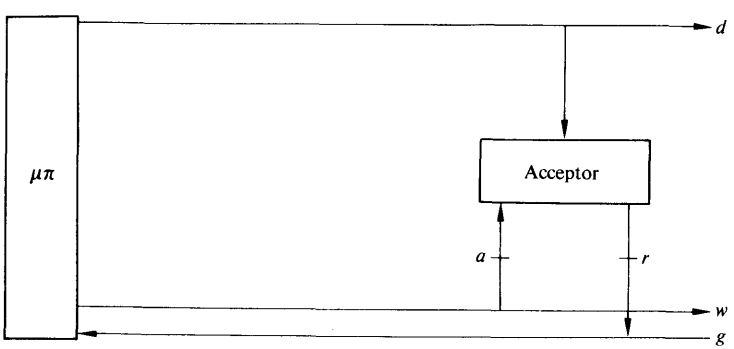


Figure 3.7.

In the case of an action/status source, some additional hardware may be required to generate the I/O port enable signal e, as discussed below.

Clearly I/O synchronization with a data source can be achieved by the microprocessor entering the wait state either during or after the I/O execute cycle, as shown in Figure 3.8. Signal 'read' equals 1 during the period that information from a peripheral is being read into the microprocessor.

In the first case, illustrated in Figure 3.8 (a), an I/O input operation is implemented by first activating the device and then reading. For example 'advance tape and read', whereas in the second case shown in Figure 3.8 (b) the opposite is true; that is, the data is first read and the device then activated. For example 'read and advance tape'.

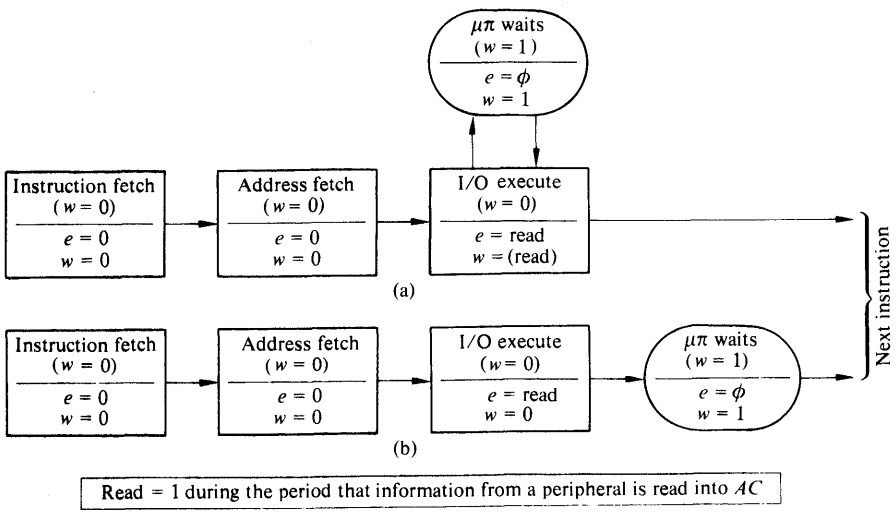


Figure 3.8.

By direct reference to Figure 3.8 (a), we obtain

$$e = \text{read} + (\text{wait}), \text{ and}$$
$$w = \text{wait} + (\text{read}).$$

Therefore,

$$e = \text{wait} + \text{read}$$
$$= w \qquad 3\ (a)$$

See Figure 3.9(a).

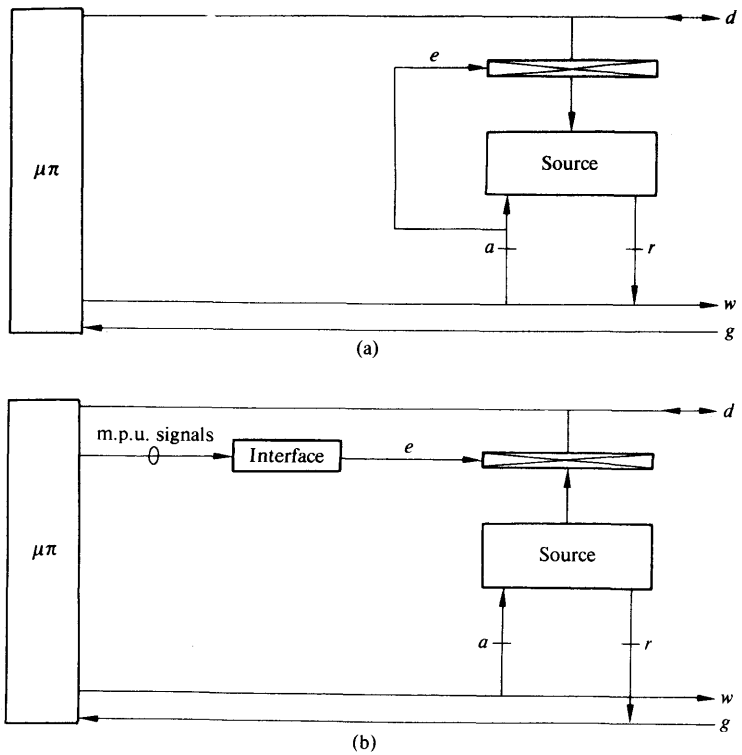


Figure 3.9.

Similarly, by reference to Figure 3.8 (b), we obtain

$$\begin{aligned} e &= \text{read} + (\text{wait}) \\ &= \text{read} \\ w &= \text{wait} \end{aligned} \qquad 3\ (b)$$

This indicates that in addition to the two wires, when the microprocessor enters a wait state at the end of an instruction, as shown in Figure 3.8(b), some simple logic is required to generate the enable signal of the tristate, e—see Figure 3.9(b). The Motorola 6800 belongs to this category unlike the INTEL 8080 which belongs to the first category.

To expand our system to accommodate n devices we modify the above equations (1, 2 and 3) to those shown below.

$$\begin{aligned} a_m &= A_m \cdot w \\ g &= A_0 \cdot r_0 + A_1 \cdot r_1 + \ldots + A_{n-1} \cdot r_{n-1} \\ e_m &= A_m \cdot e \end{aligned}$$

Their implementation is shown in Figure 3.10. In Figure 3.11 we show a simplified representation of a wait/go system. Clearly the wait/go address must be maintained when the microprocessor is in a wait state.

Note that while digital information travels from one source to one (or more) acceptors, all other sources that are tied to the data bus must be disconnected from it—that is their I/O ports must be disabled. Acceptors on the other hand can be connected directly to the data bus.

In our system, a multi-action device, such as a cassette, will be allocated the appropriate number of wait/go slots. For example, wait/go slot 6 for 'data reverse', wait/go slot 9 for 'rewind' and so on.

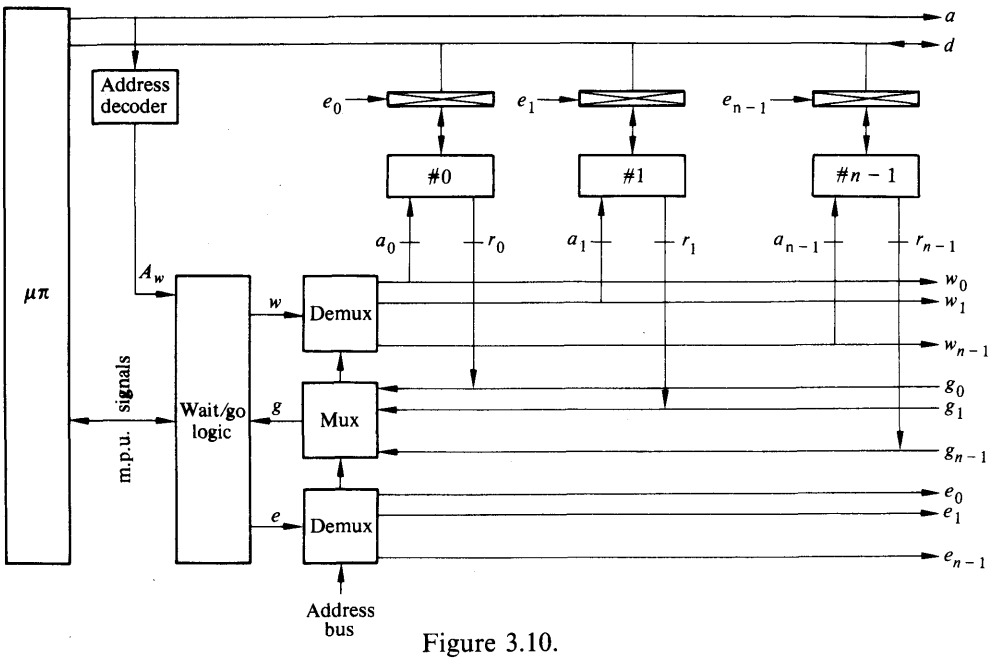


Figure 3.10.

3.4 WAIT/GO LOGIC

Although present-day microprocessors are not designed to operate in the wait/go mode, they can be made to do so by means of a relatively simple logic circuit, the *wait/go logic*, the block diagram of which is shown in Figure 3.12. Its function is to look for I/O instructions with wait/go addresses, denoted by A_w and to put the microprocessor automatically into a wait state when such an instruction is detected. Signals w, g and e have been defined in the previous section.

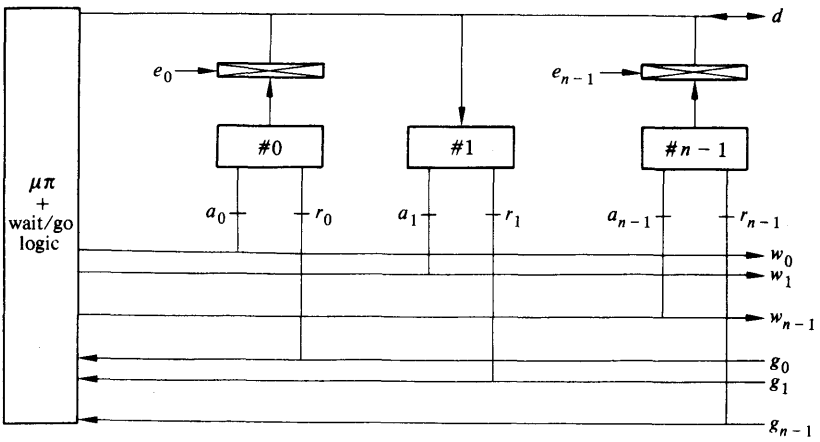


Figure 3.11.

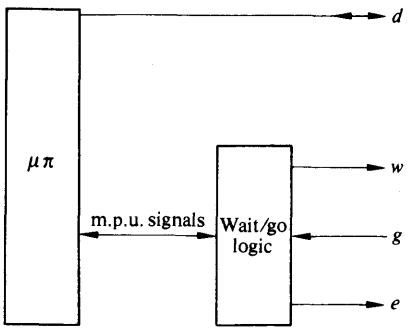


Figure 3.12.

The design and implementation of wait/go logic is straightforward and should present no difficulty to the user who possesses a working knowledge of logic design [3]. The main difficulty likely to be experienced by the designer is the correct interpretation of the m.p.u. signals and of their timing constraints.

We shall demonstrate the steps by means of the following examples

Example 1 Wait/go logic for the INTEL 8080

A set of relevant m.p.u. signals, derived from Figure 2.9, is shown in Figure 3.13(a). Their timing diagram is displayed in Figure 3.14. The state diagram of a suitable circuit is shown in Figure 3.15. It operates as follows.

The *normal state* of the circuit is S_0. This state is maintained while the microprocessor is active and all the peripherals using the wait/go mode are inactive. In this state the circuit is looking for an I/O instruction ('IN' 1 1 0 1 1 0

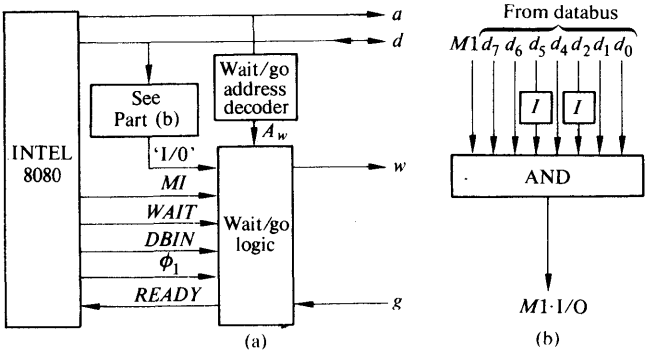


Figure 3.13.

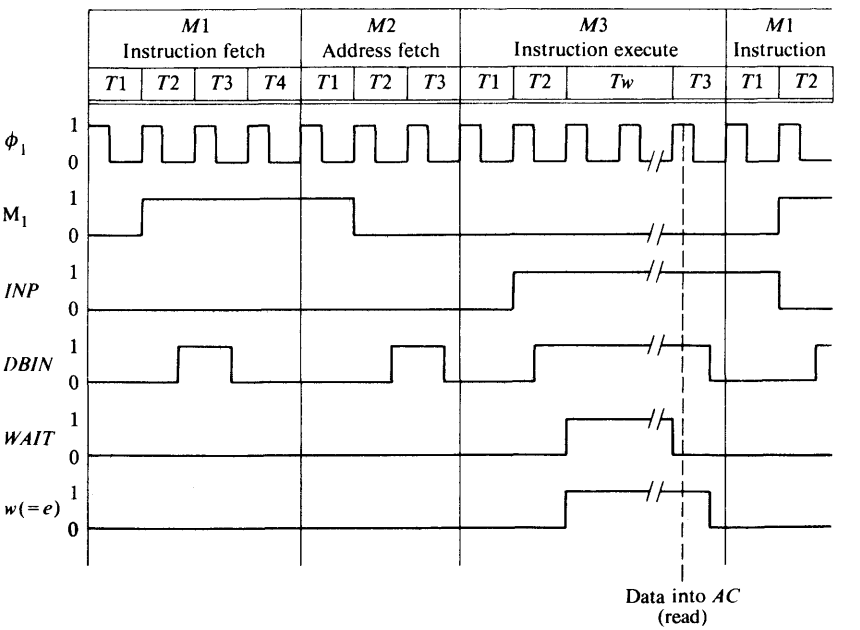


Figure 3.14.

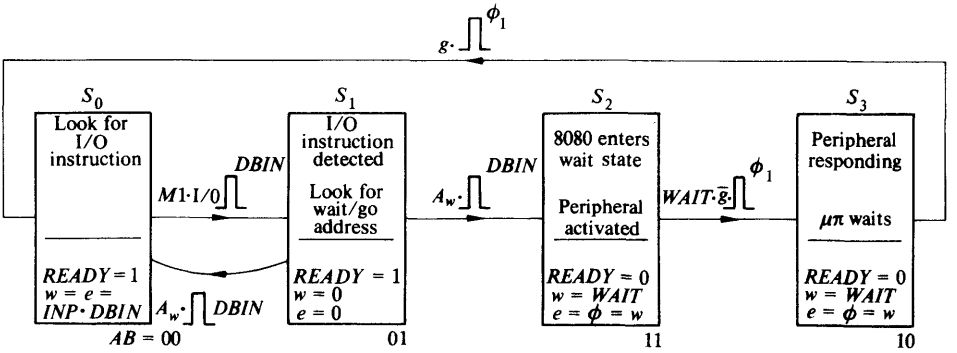


Figure 3.15.

1 1 or 'OUT' 1 1 0 1 0 0 1 1) on the data bus on its way to the m.p.u.. For this purpose we use the AND gate in Figure 3.13(b),which generates a '1' output when an I/O instruction is detected. We use the output of the AND gate to move to state S_1. State S_2 is entered one machine cycle later, that is when the microprocessor is in state $M_2 \cdot T_3$ in Figure 2.7. In this state (S_2) we pull the READY line in Figure 3.13 low. This causes the microprocessor to enter wait state $M_3 \cdot T_w$ three clock pulses later, making WAIT and w signals in Figure 3.13(a) equal to 1. Now, signal w becoming 1 activates the peripheral whose address appears on the data bus. When the peripheral responds, causing ready signal r, and therefore signal g (see Figure 3.7), to change to 0, our circuit moves to state S_3. The transition back to state S_0 takes place with the first ϕ_1 clock pulse after the peripheral activity is completed, that is after the peripheral has fully responded, indicated by signal r, and therefore signal g, changing to 1.

The implementation of the state diagram is straightforward. The steps we use to implement it have been explained in section 10 of Chapter 1. By direct reference to the state diagram in Figure 3.15, we obtain

$$\begin{aligned} S_A &= S_1 \cdot A_w \\ &= \bar{A} \cdot B \cdot A_w, \qquad \text{therefore } J_A = B \cdot A_w \end{aligned}$$

$$\begin{aligned} R_A &= S_3 \cdot g + (S_0) \\ &= A \cdot \bar{B} \cdot g + (\bar{A} \cdot \bar{B}) \\ &= \bar{B} \cdot g, \qquad \text{therefore } K_A = \bar{B} \cdot \text{g} \end{aligned}$$

$$\begin{aligned} S_B &= S_0 \cdot M1 \cdot \text{I/O} + (S_1 \cdot A_w) + (S_2 \cdot M1) + (S_3 \cdot M1) \\ &= \bar{A} \cdot \bar{B} \cdot M1 \cdot \text{I/O} + (\bar{A} \cdot B \cdot A_w) + (A \cdot B \cdot M1) + (A \cdot \bar{B} \cdot M1) \\ &= \bar{B} \cdot M1 \cdot \text{I/O}, \qquad \text{therefore } J_B = M1 \cdot \text{I/O} \end{aligned}$$

$$R_B = S_2 \cdot \text{WAIT} \cdot \bar{g} + S_1 \cdot \bar{A}_w + (S_3)$$
$$= A \cdot B \cdot \text{WAIT} \cdot \bar{g} + \bar{A} \cdot B \cdot A_w + (A\bar{B})$$
$$= A \ \text{WAIT} \cdot \bar{g} + \bar{A} \cdot B \cdot \bar{A}_w, \qquad \textit{therefore } K_B = A \cdot \text{WAIT} \cdot \bar{g} + \bar{A} \cdot \bar{A}_w$$
$$c = (S_0 + S_1) \cdot DBIN + (S_2 + S_3) \cdot \phi_1$$
$$= \bar{A} \cdot DBIN + A \cdot \phi_1$$
$$\text{READY} = S_0 + S_1$$
$$= \bar{A} \cdot \bar{B} + \bar{A} \cdot B$$
$$= \bar{A}$$

Because the INTEL 8080 enters its I/O wait state during the execute cycle of the instructions, as we explained earlier in section 3.4,

$$e = \text{wait} + \text{read}$$
$$= w$$

Now, for the INTEL 8080,

$$\text{wait} = (S_2 + S_3) \cdot WAIT = A \cdot WAIT, \text{ and}$$
$$\text{read} = INP \cdot DBIN$$

Therefore, substituting the above expressions in our equation of the enable signal, we obtain

$$e = A \cdot WAIT + INP \cdot DBIN$$
$$= w$$

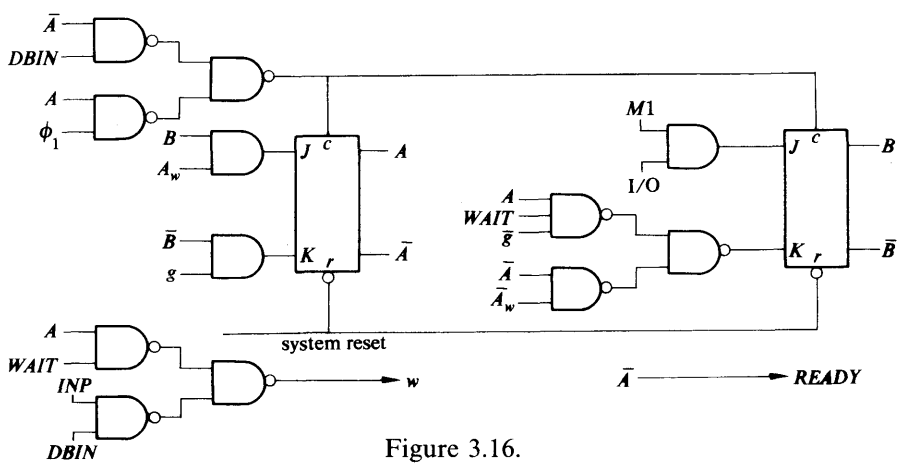


Figure 3.16.

The corresponding circuit is shown in Figure 3.16.

The reader's attention is drawn to the fact that no hardware is required to enable I/O ports in the case of wait/go systems using the INTEL 8080.

Example 2 Wait/go logic for the MOTOROLA 6800

The M6800 is halted at the end of an instruction by pulling its $\overline{HALT}$ line (pin 2 in Figure 2.10) within 100 nsecs of the leading edge of clock pulse ϕ_1 in the last cycle (see page 4.13 of M6800 Microprocessor Applications Manual 1975, reproduced in this book as Figure 3.17)

Now, as I/O operations are not discriminated from memory fetch cycles, the system designer usually allocates a block of memory addresses to I/O devices. This provides one with the opportunity of determining I/O cycles by looking either at the data bus during an address fetch operation or at the address bus during an execute cycle. Reference to relevant timing diagrams in various publications failed to provide us with a set of signals that would allow us to monitor the data bus during an address fetch cycle.† Therefore the second choice was adopted, namely monitoring the address bus during the last cycle of an instruction. The signals we used are shown in Figure 3.17—they appear on page 4.14 of the M6800 Microprocessor Applications Manual, 1975. Reference to this diagram indicates that the signals on the address bus during the last cycle become stable at a time which is closer to the trailing edge of clock pulse ϕ_1. This implies that we cannot pull its $\overline{HALT}$ line to ground within the 100 nsecs period specified by the manufacturer. The problem can be overcome in practice by inserting a 'no op' operation after each I/O instruction. Below we show two circuit implementations of wait/go logic using this method.

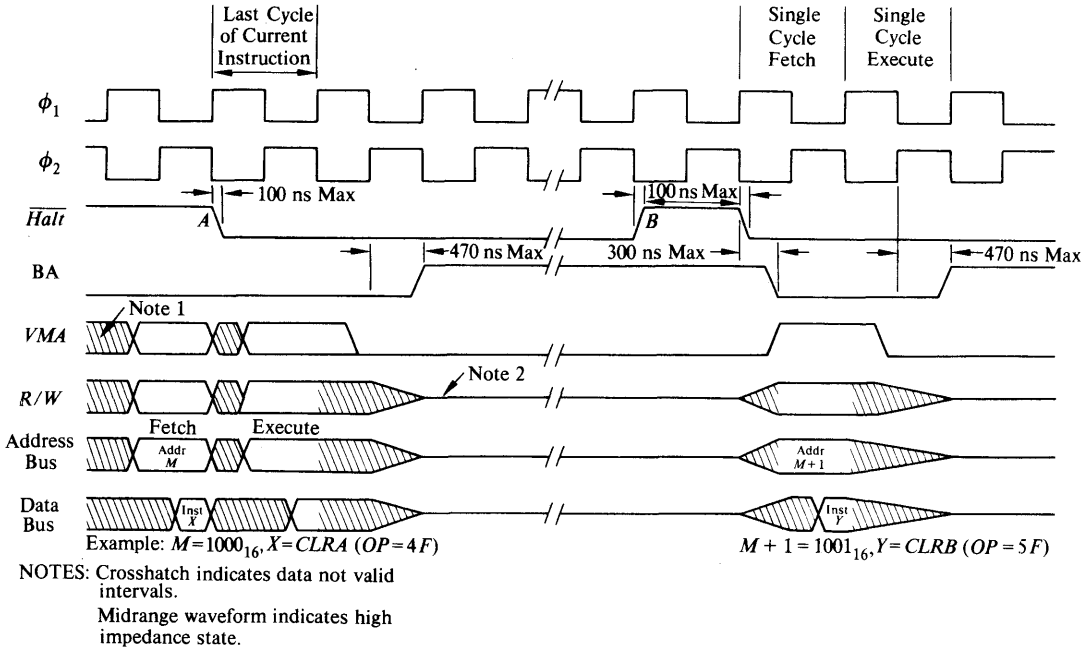
Circuit 1.

The set of the m.p.u. signals we used to implement our first wait/go circuit is shown in Figure 3.18. Their timing diagram is displayed in Figure 3.17. The state diagram of a suitable circuit is shown in Figure 3.19. Its operation is as follows.

Its 'normal' state is S_0. This state is maintained while the microprocessor is active and all the peripherals using the wait/go mode are inactive. When a wait/go address is detected on the address bus during the last cycle of the current instruction, our circuit moves to state S_1 with clock pulse ϕ_2. Note that reference to Figure 3.17 shows that both VMA and address signals are stable during ϕ_2 of the last cycle of the current instruction.

In this state we pull the microprocessor's HALT line (pin 2 in Figure 2.10) low. This forces the M6800 to enter its wait state (halt) state at the end of the next (no op) instruction. When it enters its halt state *BA*, the bus available

†The author does not wish to imply that such a signal set does not necessarily exist.



NOTES: Crosshatch indicates data not valid intervals.
Midrange waveform indicates high impedance state.

Figure 3.17.

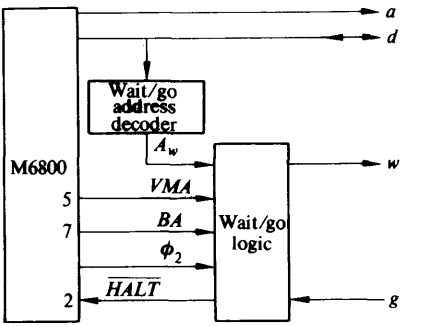


Figure 3.18.

signal, becomes 1, as shown in Figures 2.8 and 2.10†. This causes our wait signal, w, in Figure 3.7 to change to 1, which in turn activates the peripheral in question. When the peripheral responds, causing its ready signal r, and therefore signal g in Figure 3.7, to change to 0, our circuit moves to state S_2.

The transition to state S_0 takes place with the first ϕ_2 clock pulse after the peripheral has fully responded, indicated by signal r and therefore signal g in Figure 3.7, changing to 1.

The steps we use to implement our state diagram have been explained in section 9 of Chapter 1. Applying these steps and referring to our state diagram in Figure 3.19, we obtain‡

$$S_A = S_1 \cdot \bar{g} \cdot \text{`}BA\text{'}$$
$$= \bar{A} \cdot B \cdot \bar{g} \cdot \text{`}BA\text{'}, \qquad \text{therefore } J_A = B \cdot \bar{g} \cdot \text{`}BA\text{'}$$

$$R_A = S_2 \cdot g + S_3$$
$$= A \cdot B \cdot g + A \cdot \bar{B}$$
$$= A \cdot g + A \cdot \bar{B}, \qquad \text{therefore } K_A = g + \bar{B}$$

$$S_B = S_0 \cdot VMA \cdot A_w$$
$$= \bar{A} \cdot \bar{B} \cdot VMA \cdot A_w \qquad \text{therefore } J_B = \bar{A} \cdot VMA \cdot A_w$$

$$R_B = S_2 \cdot g$$
$$= A \cdot B \cdot g, \qquad \text{therefore } K_B = A \cdot g$$

$$\overline{\text{HALT}} = S_0 + S_3$$
$$= \bar{A}\bar{B} + A\bar{B} = \bar{B}$$

$$w = S_1 \cdot \text{`}BA\text{'} + S_2$$
$$= \bar{A} \cdot B \cdot \text{`}BA\text{'} + A \cdot B$$
$$= B \cdot \text{`}BA\text{'} + A \cdot B$$

†Note that the data, address and R/W lines are tristated when the microprocessor is halted.
‡We shall use variable A_w to denote wait/go addresses.

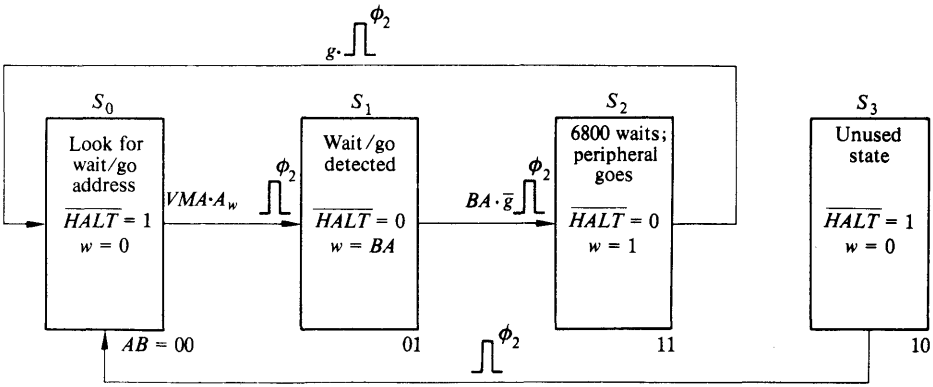


Figure 3.19.

In the case of the MOTOROLA 6800,

$$\text{read} = \text{VMA} \cdot \text{A}_w \cdot \phi_2 \cdot \text{R/W} \qquad \text{see Figure 3.17}$$

Substituting this value in equation 3(b) on page 61, we obtain

$$\begin{aligned} e &= \text{read} \\ &= VMA \cdot A_w \cdot \phi_2 \cdot RW \end{aligned} \qquad \text{see Figure 3.17}$$

The corresponding circuit is shown in Figure 3.20.

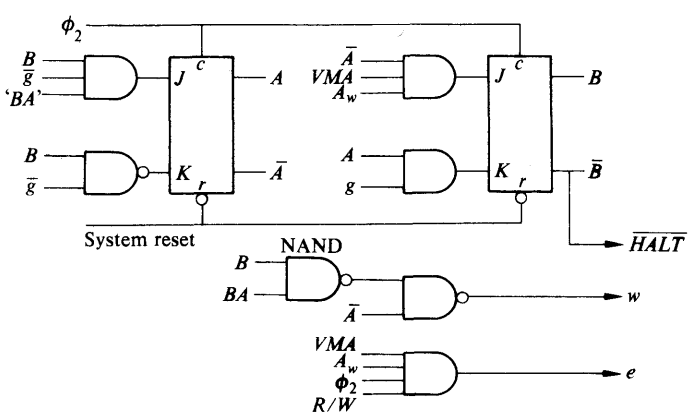


Figure 3.20.

Circuit 2.

The state diagram of an alternative wait/go logic circuit for the M6800 is shown in Figure 3.21. Its operation is self-explanatory.

Observe that whereas the previous circuit is synchronous (clock-driven), this circuit is asynchronous (event-driven). Asynchronous circuits have been discussed in section 9 of Chapter 1.

Its Boolean equations, derived directly from the state diagram, are

turn-on set of $A = B \cdot \text{`}BA\text{'}$
turn-off set of $A = \bar{B} \cdot \text{`}\overline{BA}\text{'} \cdot \phi_2 \xrightarrow{\text{Invert}} B + \text{`}BA\text{'} + \bar{\phi}_2$
turn-on set of $B = \bar{A} \cdot \text{`}VMA\text{'} \cdot A_w \cdot \phi_2$
turn off set of $B = A \cdot \bar{g} \xrightarrow{\text{Invert}} \bar{A} + g$

Therefore the circuit equations are

$$A = B \cdot \text{`}BA\text{'} + A \cdot (B + \text{`}BA\text{'} + \bar{\phi}_2)$$

$$B = \bar{A} \cdot \text{`}VMA\text{'} \cdot A_w \cdot \phi_2 + B(\bar{A} + g)$$

$$\begin{aligned} \overline{HALT} &= S_0 + S_3 \cdot g \\ &= \bar{A} \cdot \bar{B} + A \cdot \bar{B} \cdot g \\ &= \bar{A} \cdot \bar{B} + \bar{B} \cdot g \end{aligned}$$

$$\begin{aligned} w &= S_2 + S_3 \\ &= A \cdot B + A \cdot \bar{B} \\ &= A \end{aligned}$$

As for circuit 1

$$\begin{aligned} e &= \text{read} \\ &= VMA \cdot A_w \cdot \phi_2 \cdot R/W \end{aligned}$$

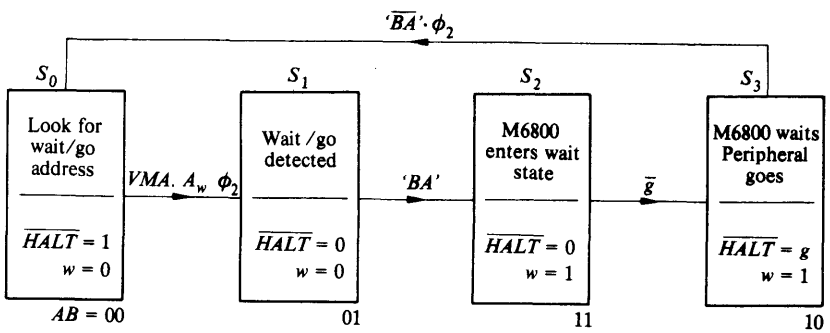


Figure 3.21.

The corresponding circuit is shown in Figure 3.22. To maintain clarity of design we have not attempted to reduce the equations.

Reference to Figure 2.10 shows that a certain interplay exists between the 'bus available' signal BA and the tristate control input $TSC - BA$ is forced low when $TSC = 1$. This will clearly cause us to lose our w signal, if TSC is applied while the microprocessor is executing a wait/go cycle. This problem can be overcome by disabling pin 39 in Figure 2.10 during each wait/go cycle. We do so by ANDing the TSC input with $\overline{S_2 + S_3}$. That is

$$\begin{aligned} \text{pin } 39 &= \overline{S_2 + S_3} \cdot TSC \\ &= (A \cdot B + A \cdot \bar{B}) TSC \\ &= \bar{A} \cdot TSC \end{aligned}$$

States S_2 and S_3 appear in Figure 3.21.

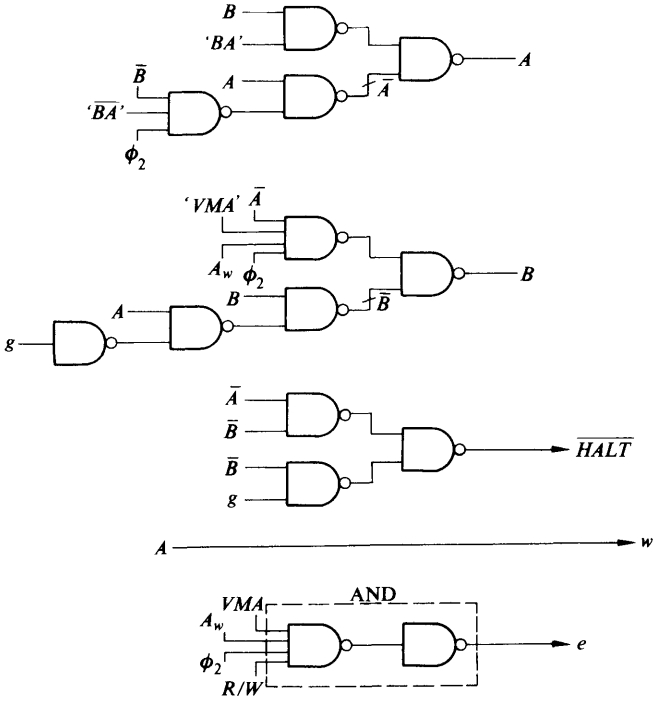


Figure 3.22.

3.5 PROBLEMS AND SOLUTIONS

In this section we shall demonstrate the design procedures by means of problems and solutions. The reader's attention is drawn to the fact that, although we use the INTEL 8080 and the MOTOROLA 6800 to implement our designs, our procedures apply to all types of microprocessors. Specifically it should be noted that the first three steps in the design are executed without reference to the microprocessor.

Problem 1 *Search for a record*

Given a paper tape reader and a microprocessor, design a system that stops the tape and raises a flag when the character sequence 4–5–6 is detected.

Use the wait/go mode to implement your design, which is to be verified using the INTEL 8080 and the MOTOROLA 6800.

8080 SOLUTION

Step 1 *Aim of the design*

The main aim is to scan incoming data for specified sequences, such as labels, threshold values, etc.

Step 2 *Device characteristics*

The microprocessor has wait/go logic, and the reader is an action/status device.† The tape is an ASCII tape—see Figure 3.23.

Step 3 *System design*

The block diagram of our solution is shown in Figure 3.24. Its step-by-step operation is flow-charted in Figure 3.25.

Step 4 *Hardware design*

With the exception of the I/O port in Figure 3.26 no other hardware is needed. This is because for the INTEL 8080 $e = w$. This was proved in section 3.4.

Step 5 *Software design*

The octal and hexadecimal listings for the INTEL 8080 are derived by direct reference to the flow chart in Figure 3.25 and to the program chart in Figure 3.27 (or to the instruction set in Figure 3.28).

---

†Action/status devices are explained in Appendix 1.

ASCII Character Codes

Character	*Octal code:* 7 *bit*	8 *bit*	*Character*	*Octal code:* 7 *bit*	8 *bit*	*Character*	*Octal code.* 7 *bit*	8 *bit*
(space)	040	240	0	060	260	*H*	110	310
			1	061	261	*I*	111	311
"	042	242	2	062	262	*J*	112	312
#	043	243	3	063	263	*K*	133	333
$	044	244	4	064	264	*L*	114	314
%	045	245	5	065	265	*M*	115	315
&	046	246	6	066	266	*N*	116	316
' (quote)	047	247	7	067	267	*O*	117	317
(	050	250	8	070	270	*P*	120	320
)	051	251	9	071	271	*Q*	121	321
*	052	252	:	072	272	*R*	122	322
+	053	253	;	073	273	*S*	123	323
, (comma)	054	254				*T*	124	324
-	055	255	=	075	275	*U*	125	325
.	056	256	?	077	277	*V*	126	326
/	057	257	@	100	300	*W*	127	327
line feed	012	212	*A*	101	301	*X*	130	330
carriage rt	015	215	*B*	102	302	*Y*	131	331
			C	103	303	*Z*	132	332
			D	104	304			
			E	105	305			
			F	106	306			
null	177	377	*G*	107	307			

Figure 3.23.

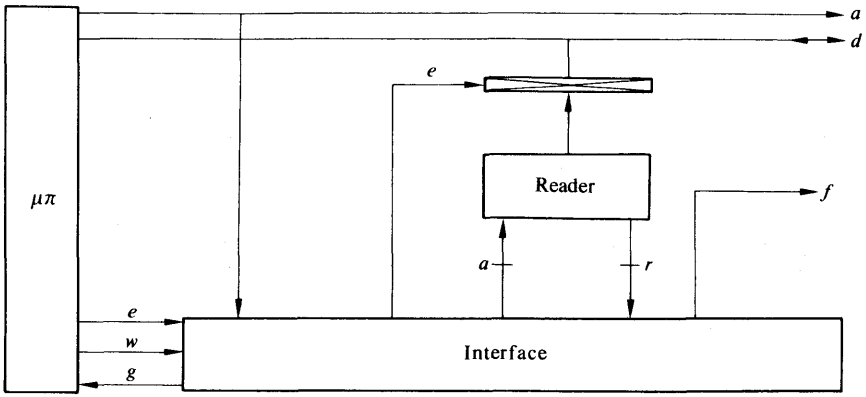


Figure 3.24.

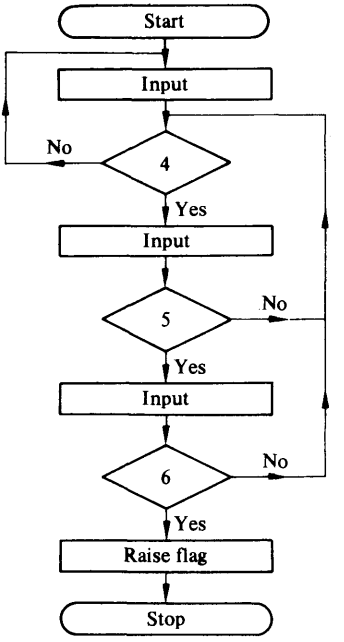


Figure 3.25.

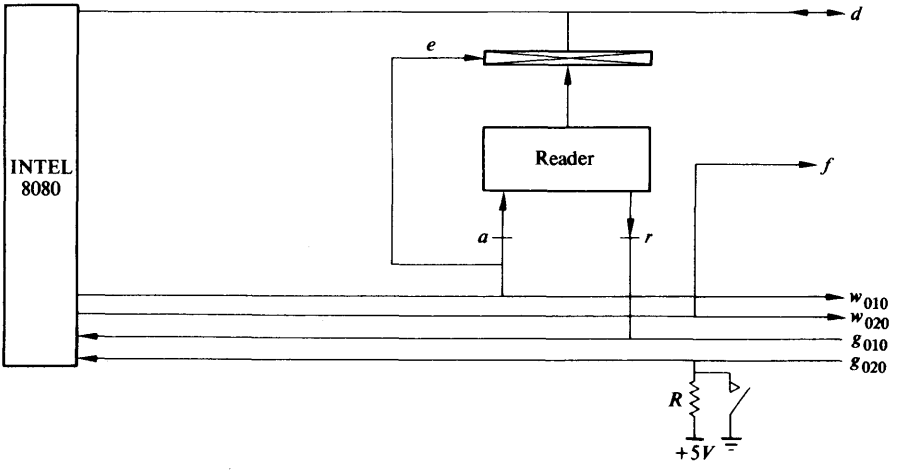


Figure 3.26.

			D6 = 0								D6 = 1								
			$D_2D_1D_0$								$D_2D_1D_0$								
			000	010	110	100	101	111	011	001	001	011	111	101	100	110	010	000	
		000	NOP 000	Mp = A 002	006	004	005	ROTATE 007	003	001	101	103	107 r1: = r2	105	104	r1: = Mhl 106	102 r1: = r2	100	B
		010		022	026	024	025	027	P+1 023	P: = H 021	121	123	127	125	124	r1: = Mhl 126	122	120	D
		110		M1: = A 062	066	064	065	c: = 1 067	063	061	161	163	167	165	164	HALT 166	162	160	
		100		M1: = HL 042	r: = r+1 046	044	r: = r−1 045	BCD A: 047	043	041	141	143	147	145	144	146	142	140	H
0	D5	D4 D3		HL: = M1 052	056	054	055	A: = Ā 057	053	051	151	153	157 r1: = r2	155	154	r1: = Mhl 156	152 r1: = r2	150	L
		111		A: = M1 072	076	074	075	c: = c̄ 077	P−1 073	071	171	173	177	175	174	176	172	170	A
		011		A: = Mp 032	036	034	035	ROTATE 037	P: = P−1 033	HL: = HL+P 031	131	133	137	135	134	r1: = Mhl 136	132	130	E
		001		A: 012	016	014	015	017	013	011	111	113	117	115	114	116	112	110	C
D_7																			
	+c	001	210	212	216	214	215	217	213	211	RET 311		317	CALL 315	314	316	312	310	z
	−c	011	230	232	236	234	235	237	233	231		INPUT 333	337		334	336	332	330	c
	(-)	111	270	272	276	274	275	277	273	271	SP: = HL 371	EI 373	377		374	376	372	370	n
	⊻	101	250	252	256	254	255	257	253	251	PC: = HL 351	353	357		354	356	352	350	p
1	D5	D4 D3				A: = A op r										A: = A op I			
	∧	100	240	242	246	244	245	247	243	241	341	343	RESTART 347	345	344	346	342	340	p̄
	∨	110	260	262	266	264	265	267	263	261	POP P 361	DI 363	367	PUSH P 365	364	366	362	360	n̄
	−	010	220	222	226	224	225	227	223	221	311	OUT 323	327	325	324	326	322	320	c̄
	+	000	200	202	206	204	205	207	203	201	301	JUMP 303	307	305	304	306	302	300	z̄
															CONDITIONAL CALL		CONDITIONAL JUMP	CONDITIONAL RETURN	

	Single Register r	Condition	Operator op	Double Register P		Rotate (A)
000	B	z̄ non zero	+	00	BC	Left Logical
010	D	c̄ no carry	−	01	DE	Right Logical
110	Mhl	n̄ pos or zero	∨	11	SP*	Right Arithmetic
100	H	p̄ odd parity	∧	10	HL	Left Arithmatic
101	L	p even parity	⊻			
111	A	n negative	compare (z: = A = r; c: = A < r)			
011	E	c carry	−c (A: = A − r − c)			
001	C	z zero	+c (A: = A − r + c)			

*AF in Pop, Push

Program chart for INTEL 8080 (Developed by F. G. Duncan, University of Bristol, England).

Figure 3.27.

Mnemonic	Description	Instruction Code (1) D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Clock (2) Cycles
ACI	Add immediate to *A* with carry	1	1	0	0	1	1	1	0	7
ADC M	Add memory to *A* with carry	1	0	0	0	1	1	1	0	7
ADC r	Add register to *A* with carry	1	0	0	0	1	*S*	*S*	*S*	4
ADD M	Add memory to *A*	1	0	0	0	0	1	1	0	7
Add r	Add register to *A*	1	0	0	0	0	*S*	*S*	*S*	4
ADI	Add immediate to *A*	1	1	0	0	0	1	1	0	7
ANA M	And memory with *A*	1	0	1	0	0	1	1	0	7
ANA r	And register with *A*	1	0	1	0	0	*S*	*S*	*S*	4
ANI	And immediate with *A*	1	1	1	0	0	1	1	0	7
CALL	Call unconditional	1	1	0	0	1	1	0	1	17
CC	Call on carry	1	1	0	1	1	1	0	0	11/17
CM	Call on minus	1	1	1	1	1	1	0	0	11/17
CMA	Complement *A*	0	0	1	0	1	1	1	1	4
CMC	Complement carry	0	0	1	1	1	1	1	1	4
CMP M	Compare memory with *A*	1	0	1	1	1	1	1	0	7
CMP r	Compare register with *A*	1	0	1	1	1	*S*	*S*	*S*	4
CNC	Call on no carry	1	1	0	1	0	1	0	0	11/17
CNZ	Call on no zero	1	1	0	0	0	1	0	0	11/17
CP	Call on positive	1	1	1	1	0	1	0	0	11/17
CPE	Call on parity even	1	1	1	0	1	1	0	0	11/17
CPI	Compare immediate with *A*	1	1	1	1	1	1	1	0	7
CPO	Call on parity odd	1	1	1	0	0	1	0	0	11/17
CZ	Call on zero	1	1	0	0	1	1	0	0	11/17
DAA	Decimal adjust *A*	0	0	1	0	0	1	1	1	4
DAD B	Add *B* & *C* to *H* & *L*	0	0	0	0	1	0	0	1	10
DAD D	Add *D* & *E* to *H* & *L*	0	0	0	1	1	0	0	1	10
DAD H	Add *H* & *L* to *H* & *L*	0	0	1	0	1	0	0	1	10
DAD SP	Add stack pointer to *H* & *L*	0	0	1	1	1	0	0	1	10
DCR M	Decrement memory	0	0	1	1	0	1	0	1	10
DCR r	Decrement register	0	0	*D*	*D*	*D*	1	0	1	5
DCX B	Decrement *B* & *C*	0	0	0	0	1	0	1	1	5
DCX D	Decrement *D* & *E*	0	0	0	1	1	0	1	1	5
DCX H	Decrement *H* & *L*	0	0	1	0	1	0	1	1	5
DCX SP	Decrement stack pointer	0	0	1	1	1	0	1	1	5
DI	Disable interrupt	1	1	1	1	0	0	1	1	4
EI	Enable Interrupts	1	1	1	1	1	0	1	1	4
HLT	Halt	0	1	1	1	0	1	1	0	7
IN	Input	1	1	0	1	1	0	1	1	10
INR M	Increment memory	0	0	1	1	0	1	0	0	10
INR r	Increment register	0	0	*D*	*D*	*D*	1	0	0	5
INX B	Increment *B* & *C* registers	0	0	0	0	0	0	1	1	5
INX D	Increment *D* & *E* registers	0	0	0	1	0	0	1	1	5
INX H	Increment *H* & *L* registers	0	0	1	0	0	0	1	1	5
INX SP	Increment stack pointer	0	0	1	1	0	0	1	1	5
JC	Jump on carry	1	1	0	1	1	0	1	0	10
JM	Jump on minus	1	1	1	1	1	0	1	0	10
JMP	Jump unconditional	1	1	0	0	0	0	1	1	10
JNC	Jump on no carry	1	1	0	1	0	0	1	0	10
JNZ	Jump on no zero	1	1	0	0	0	0	1	0	10
JP	Jump on positive	1	1	1	1	0	0	1	0	10
JPE	Jump on parity even	1	1	1	0	1	0	1	0	10
JPO	Jump on parity odd	1	1	1	0	0	0	1	0	10
JZ	Jump on zero	1	1	0	0	1	0	1	0	10
LDA	Load *A* direct	0	0	1	1	1	0	1	0	13
LDAX B	Load *A* indirect	0	0	0	0	1	0	1	0	7
LDAX D	Load *A* indirect	0	0	0	1	1	0	1	0	7
LHLD	Load *H* & *L* direct	0	0	1	0	1	0	1	0	16
LXI B	Load immediate register Pair *B* & *C*	0	0	0	0	0	0	0	1	10
LXI D	Load immediate register Pair *D* & *E*	0	0	0	1	0	0	0	1	10
LXI H	Load immediate register Pair *H* & *L*	0	0	1	0	0	0	0	1	10
LXI SP	Load immediate stack pointer	0	0	1	1	0	0	0	1	10

Mnemonic	Description	Instruction Code (1) D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Clock (2) Cycles
MVI M	Move immediate memory	0	0	1	1	0	1	1	0	10
MVI r	Move immediate register	0	0	*D*	*D*	*D*	1	1	0	7
MOV M,r	Move register to memory	0	1	1	1	0	*S*	*S*	*S*	7
MOV r, M	Move memory to register	0	1	*D*	*D*	*D*	1	1	0	7
MOV r1, r2	Move register to register	0	1	*D*	*D*	*D*	*S*	*S*	*S*	5
NOP	No-operation	0	0	0	0	0	0	0	0	4
ORA M	Or memory with *A*	1	0	1	1	0	1	1	0	7
ORA r	Or register with *A*	1	0	1	1	0	*S*	*S*	*S*	4
ORI	Or immediate with *A*	1	1	1	1	0	1	1	0	7
OUT	Output	1	1	0	1	0	0	1	1	10
PCHL	*H* & *L* to program counter	1	1	1	0	1	0	0	1	5
POP B	Pop register pair *B* & *C* off stack	1	1	0	0	0	0	0	1	10
POP D	Pop register pair *D* & *E* off stack	1	1	0	1	0	0	0	1	10
POP H	Pop register pair *H* & *L* off stack	1	1	1	0	0	0	0	1	10
POP PSW	Pop *A* and Flags off stack	1	1	1	1	0	0	0	1	10
PUSH B	Push register Pair *B* & *C* on stack	1	1	0	0	0	1	0	1	11
PUSH D	Push register Pair *D* & *E* on stack	1	1	0	1	0	1	0	1	11
PUSH H	Push register Pair *H* & *L* on stack	1	1	1	0	0	1	0	1	11
PUSH PSW	Push *A* and Flags on stack	1	1	1	1	0	1	0	1	11
RAL	Rotate *A* left through carry	0	0	0	1	0	1	1	1	4
RAR	Rotate *A* right through carry	0	0	0	1	1	1	1	1	4
RC	Return on carry	1	1	0	1	1	0	0	0	5/11
RET	Return	1	1	0	0	1	0	0	1	10
RLC	Rotate *A* left	0	0	0	0	0	1	1	1	4
RM	Return to minus	1	1	1	1	1	0	0	0	5/11
RNC	Return on no carry	1	1	0	1	0	0	0	0	5/11
RNZ	Return on no zero	1	1	0	0	0	0	0	0	5/11
RP	Return on positive	1	1	1	1	0	0	0	0	5/11
RPE	Return on parity even	1	1	1	0	1	0	0	0	5/11
RPO	Return on parity odd	1	1	1	0	0	0	0	0	5/11
RRC	Rotate *A* right	0	0	0	0	1	1	1	1	4
RST	Restart	1	1	*A*	*A*	*A*	1	1	1	11
RZ	Return on zero	1	1	0	0	1	0	0	0	5/11
SBB M	Subtract memory from *A* with borrow	1	0	0	1	1	1	1	0	7
SBB r	Subtract register from *A* with borrow	1	0	0	1	1	*S*	*S*	*S*	4
SBI	Subtract immediate from *A* with borrow	1	1	0	1	1	1	1	0	7
SHLD	Store *H* & *L* direct	0	0	1	0	0	0	1	0	16
SPHL	*H* & *L* to stack pointer	1	1	1	1	1	0	0	1	5
STA	Store *A* direct	0	0	1	1	0	0	1	0	13
STAX B	Store *A* indirect	0	0	0	0	0	0	1	0	7
STAX D	Store *A* indirect	0	0	0	1	0	0	1	0	7
STC	Set carry	0	0	1	1	0	1	1	1	4
SUB M	Subtract memory from *A*	1	0	0	1	0	1	1	0	7
SUB r	Subtract register from *A*	1	0	0	1	0	*S*	*S*	*S*	4
SUI	Subtract immediate from *A*	1	1	0	1	0	1	1	0	7
XCHG	Exchange *D* & *E*, *H* & *L* Registers	1	1	1	0	1	0	1	1	4
XRA M	Exclusive Or memory with *A*	1	0	1	0	1	1	1	0	7
XRA r	Exclusive Or register with *A*	1	0	1	0	1	*S*	*S*	*S*	4
XRI	Exclusive Or immediate with *A*	1	1	1	0	1	1	1	0	7
XTHL	Exchange top of stack, *H* & *L*	1	1	1	0	0	0	1	1	18

Figure 3.28. Instruction set for the INTEL 8080 (Reproduced from INTEL 8080 Microcomputer). Summary of Processor Instructions. By Alphabetical Order.

Notes: 1. *DDD* or *SSS*–000 *B*–001 *C*–010 *D*–011 *E*–100 *H*–101 *L*–110 Memory–111 *A*.
2. Two possible cycle times, (5/11) indicate instruction cycles dependent on condition flags.

Octal address			Octal listing	Hex listing	Mnemonics	Comments
	H	L				
L1:	003	000	333	DB	IN	Read next character.
		001	010	08		
L2:		002	376	FE	CPI	Compare AC with 4.†
		003	264	B4		
		004	302	C2	JNZ	If character not a 4, jump to L1.
		005	000	00	L1	
		006	003	03		
		007	333	DB	IN	Read next character.
		010	010	08		
		011	376	FE	CPI	Compare AC with 5.†
		012	265	B5		
		013	302	C2	JNZ	If character not a 5, jump to L2.
		014	002	02	L2	
		015	003	03		
		016	333	DB	IN	Read next character.
		017	010	08		
		020	376	FE	CPI	Compare AC with 6.†
		023	266	B6		
		022	302	C2	JNZ	If character not a 6, jump to L2.
		023	002	02	L2	
		024	003	03		
		025	323	D3	OUT	Raise flag.
		026	020	10		
		027	166	76	HLT	Halt.

† See ASCII Table on p. 74.

6800 SOLUTION

Step 1
Step 2
Step 3
} Same as in the 8080 solution

Step 4 *Hardware design*

The block diagram of our solution using the M6800 is shown in Figure 3.29. In addition to the I/O port, we require an AND gate to generate the enable signal *e*, as explained in section 3.4—see circuit 1 and circuit 2 of example 2.

Step 5 *Software design*

By direct reference to the flow chart in Figure 3.25 and to the instruction set in Figure 3.30 or to the programming chart in Figure 3.31, we obtain the hexadecimal listing of our program, which is shown on p. 79.

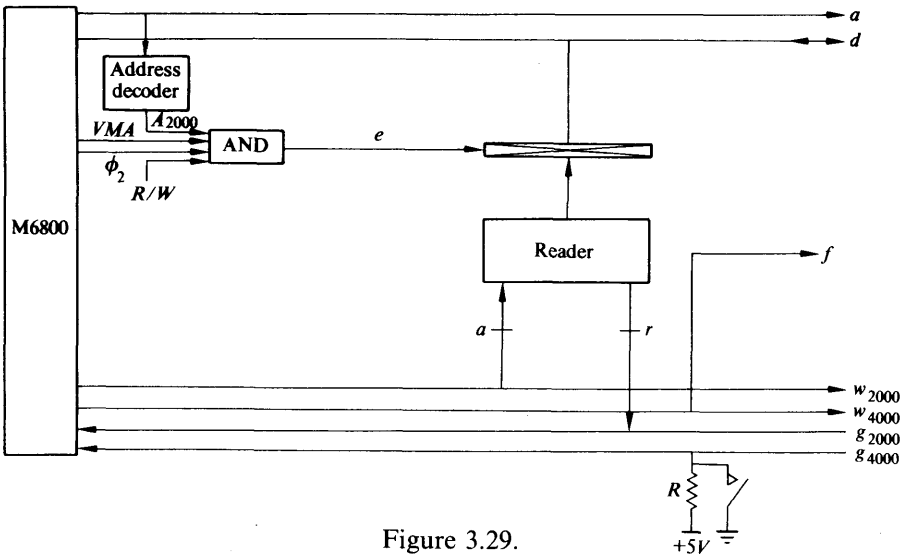


Figure 3.29.

	Hex address		*Hex listing*	*Mnemonics*	*Comments*
	H	*L*			
L1:	00	00	*B6*	*LDA A*,2000	Read next character.
		01	20		
		02	00		
		03	01	*NOP*	
L2:		04	81	*CMP A*	Compare *A* with 4.
		05	34		
		06	26	*BNE L1*	If character not a 4, jump to *L1*.
		07	*F8*		
		08	*B6*	*LDA A*,2000	Read next character.
		09	20		
		0*A*	00		
		0*B*	01	*NOP*	
		0*C*	81	*CMP A*	Compare *A* with 5.
		0*D*	35		
		0*E*	26	*BNE L2*	If character not a 5, jump to *L2*.
		0*F*	*F4*		
		10	*B6*	*LDA A*,2000	Read next character.
		11	20		
		12	00		
		13	01	*NOP*	
		14	81	*CMP A*	Compare *A* with 6.
		15	36		
		16	26	*BNE L2*	If character not a 6, jump to *L2*.
		17	*EC*		
		18	*B7*	*STA A*,4000	Raise flag.
		19	40		
		1*A*	00		
		1*B*	*3F*	*SWI*	Stop.

Figure 3.31 Program Chart for MOTOROLA 6800

D_7	$D_5 D_4 D_3$	D_6 = 0, $D_2 D_1 D_0$ = 000	010	110	100	101	111	011	001
0	000			CC:= A			A:= CC		NOP
	010	A:= A−B		B:= A			A:= B		Compare A, B
	110	IX:= SP+1	POP A	PUSH A	SP:= SP−1	SP:= IX−1	PUSH B	POP B	SP:= SP+1
	100	JUMP	$\bar{c}\bar{z}$	$\bar{z}$	$\bar{c}$	c	z	c+z	
	$D_5 D_4 D_3$	CONDITIONAL JUMP							
	101	$\bar{v}$	$\bar{n}$	$\bar{z}$ $(nv+\bar{n}\bar{v})$	$nv+\bar{n}\bar{v}$	$n\bar{v}+\bar{n}v$	$z+n\bar{v}+\bar{n}v$	n	v
	111			WAIT			INT SOFT	RET INT	RET SUBR
	011							A:= A+B	DAA
	001	IX:= IX+1	v:=0	I:=0	c:=0	c:=1	I:=1	v:=1	IX:= IX−1
1	001	A:= A∧p	A:= A∨p	SP:=p	TEST (IX−p)	SUBR JUMP	p:= SP	A:= A+p	A:= A+p+c
	011								
	111								
	101					SUBR JUMP			
	$D_5 D_4 D_3$								
	100	A:= A−p	A:= A−p−c	A:=p	A:= A∧p	TEST (A∧p)	p:= A		TEST (A−p)
	110								
	010								
	000								

D_6 = 1, $D_2 D_1 D_0$ =	001	011	111	101	100	110	010	000
	ROTATE LEFT g	$g:=\bar{g}$	ARITH SHIFT RIGHT g	TEST (g−0)	LOGIC SHIFT RIGHT g	ROTATE RIGHT g		g:= −g
			g:=0		g:=g+1	JUMP IND	g:=g−1	SHIFT LEFT g
						JUMP EXT		
	B:= B+p+c	B:= B+p	p:= IX			IX:=p	B:= B∨p	B:= B∨p
	TEST (B−p)		p:= B	TEST (B∧p)	B:= B∧p	B:=p	B:= B−p−c	B:= B−p

Operand

$D_5 D_4$	p	g
0 0	IMM	A
0 1	DIR	B
1 1	EXT	EXT
1 0	IND	IND WITH $D_6 = 1$
		OR REL WITH $D_6 = 0$

Registers

A	ACCUMULATOR (8)
B	ACCUMULATOR (8)
IX	INDEX REGISTER (16)
PC	PROGRAM COUNTER (16)
SP	STACK POINTER (16)
CC	CONDITION CODES (6)

Condition Codes

c	CARRY—BORROW
v	OVERFLOW
z	ZERO
n	NEGATIVE
i	INTERRUPT MASK
h	HALF CARRY

Accumulator and memory instructions for the Motorola M6800 Microcomputer System (Reproduced from M6800 Microcomputer System Design Data 1976–7)

ACCUMULATOR AND MEMORY OPERATIONS	MNEMONIC	ADDRESSING MODES: IMMED OP	IMMED ~	IMMED #	DIRECT OP	DIRECT ~	DIRECT #	INDEX OP	INDEX ~	INDEX #	EXTND OP	EXTND ~	EXTND #	INHER OP	INHER ~	INHER #	BOOLEAN/ARITHMETIC OPERATION (All register labels refer to contents)	COND. CODE REG. 5 H	4 I	3 N	2 Z	1 V	0 C
Add	ADDA	8B	2	2	9B	3	2	AB	5	2	BB	4	3				A + M → A	↕	•	↕	↕	↕	↕
	ADDB	CB	2	2	DB	3	2	EB	5	2	FB	4	3				B + M → B	↕	•	↕	↕	↕	↕
Add Acmltrs	ABA													1B	2	1	A + B → A	↕	•	↕	↕	↕	↕
Add with Carry	ADCA	89	2	2	99	3	2	A9	5	2	B9	4	3				A + M + C → A	↕	•	↕	↕	↕	↕
	ADCB	C9	2	2	D9	3	2	E9	5	2	F9	4	3				B + M + C → B	↕	•	↕	↕	↕	↕
And	ANDA	84	2	2	94	3	2	A4	5	2	B4	4	3				A • M → A	•	•	↕	↕	R	•
	ANDB	C4	2	2	D4	3	2	E4	5	2	F4	4	3				B • M → B	•	•	↕	↕	R	•
Bit Test	BITA	85	2	2	95	3	2	A5	5	2	B5	4	3				A • M	•	•	↕	↕	R	•
	BITB	C5	2	2	D5	3	2	E5	5	2	F5	4	3				B • M	•	•	↕	↕	R	•
Clear	CLR							6F	7	2	7F	6	3				00 → M	•	•	R	S	R	R
	CLRA													4F	2	1	00 → A	•	•	R	S	R	R
	CLRB													5F	2	1	00 → B	•	•	R	S	R	R
Compare	CMPA	81	2	2	91	3	2	A1	5	2	B1	4	3				A − M	•	•	↕	↕	↕	↕
	CMPB	C1	2	2	D1	3	2	E1	5	2	F1	4	3				B − M	•	•	↕	↕	↕	↕
Compare Acmltrs	CBA													11	2	1	A − B	•	•	↕	↕	↕	↕
Complement, 1's	COM							63	7	2	73	6	3				$\overline{M} \rightarrow M$	•	•	↕	↕	R	S
	COMA													43	2	1	$\overline{A} \rightarrow A$	•	•	↕	↕	R	S
	COMB													53	2	1	$\overline{B} \rightarrow B$	•	•	↕	↕	R	S
Complement, 2's (Negate)	NEG							60	7	2	70	6	3				00 − M → M	•	•	↕	↕	①	②
	NEGA													40	2	1	00 − A → A	•	•	↕	↕	①	②
	NEGB													50	2	1	00 − B → B	•	•	↕	↕	①	②
Decimal Adjust, A	DAA													19	2	1	Converts Binary Add. of BCD Characters into BCD Format	•	•	↕	↕	↕	③
Decrement	DEC							6A	7	2	7A	6	3				M − 1 → M	•	•	↕	↕	④	•
	DECA													4A	2	1	A − 1 → A	•	•	↕	↕	④	•
	DECB													5A	2	1	B − 1 → B	•	•	↕	↕	④	•
Exclusive OR	EORA	88	2	2	98	3	2	A8	5	2	B8	4	3				A ⊕ M → A	•	•	↕	↕	R	•
	EORB	C8	2	2	D8	3	2	E8	5	2	F8	4	3				B ⊕ M → B	•	•	↕	↕	R	•
Increment	INC							6C	7	2	7C	6	3				M + 1 → M	•	•	↕	↕	⑤	•
	INCA													4C	2	1	A + 1 → A	•	•	↕	↕	⑤	•
	INCB													5C	2	1	B + 1 → B	•	•	↕	↕	⑤	•
Load Acmltr	LDAA	86	2	2	96	3	2	A6	5	2	B6	4	3				M → A	•	•	↕	↕	R	•
	LDAB	C6	2	2	D6	3	2	E6	5	2	F6	4	3				M → B	•	•	↕	↕	R	•
Or, Inclusive	ORAA	8A	2	2	9A	3	2	AA	5	2	BA	4	3				A + M → A	•	•	↕	↕	R	•
	ORAB	CA	2	2	DA	3	2	EA	5	2	FA	4	3				B + M → B	•	•	↕	↕	R	•
Push Data	PSHA													36	4	1	A → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
	PSHB													37	4	1	B → M_{SP}, SP − 1 → SP	•	•	•	•	•	•
Pull Data	PULA													32	4	1	SP + 1 → SP, M_{SP} → A	•	•	•	•	•	•
	PULB													33	4	1	SP + 1 → SP, M_{SP} → B	•	•	•	•	•	•
Rotate Left	ROL							69	7	2	79	6	3				M	•	•	↕	↕	⑥	↕
	ROLA													49	2	1	A	•	•	↕	↕	⑥	↕
	ROLB													59	2	1	B	•	•	↕	↕	⑥	↕
Rotate Right	ROR							66	7	2	76	6	3				M	•	•	↕	↕	⑥	↕
	RORA													46	2	1	A	•	•	↕	↕	⑥	↕
	RORB													56	2	1	B	•	•	↕	↕	⑥	↕
Shift Left, Arithmetic	ASL							68	7	2	78	6	3				M	•	•	↕	↕	⑥	↕
	ASLA													48	2	1	A	•	•	↕	↕	⑥	↕
	ASLB													58	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Arithmetic	ASR							67	7	2	77	6	3				M	•	•	↕	↕	⑥	↕
	ASRA													47	2	1	A	•	•	↕	↕	⑥	↕
	ASRB													57	2	1	B	•	•	↕	↕	⑥	↕
Shift Right, Logic.	LSR							64	7	2	74	6	3				M	•	•	R	↕	⑥	↕
	LSRA													44	2	1	A	•	•	R	↕	⑥	↕
	LSRB													54	2	1	B	•	•	R	↕	⑥	↕
Store Acmltr.	STAA				97	4	2	A7	6	2	B7	5	3				A → M	•	•	↕	↕	R	•
	STAB				D7	4	2	E7	6	2	F7	5	3				B → M	•	•	↕	↕	R	•
Subtract	SUBA	80	2	2	90	3	2	A0	5	2	B0	4	3				A − M → A	•	•	↕	↕	↕	↕
	SUBB	C0	2	2	D0	3	2	E0	5	2	F0	4	3				B − M → B	•	•	↕	↕	↕	↕
Subract Acmltrs.	SBA													10	2	1	A − B → A	•	•	↕	↕	↕	↕
Subtr. with Carry	SBCA	82	2	2	92	3	2	A2	5	2	B2	4	3				A − M − C → A	•	•	↕	↕	↕	↕
	SBCB	C2	2	2	D2	3	2	E2	5	2	F2	4	3				B − M − C → B	•	•	↕	↕	↕	↕
Transfer Acmltrs	TAB													16	2	1	A → B	•	•	↕	↕	R	•
	TBA													17	2	1	B → A	•	•	↕	↕	R	•
Test, Zero or Minus	TST							6D	7	2	7D	6	3				M − 00	•	•	↕	↕	R	R
	TSTA													4D	2	1	A − 00	•	•	↕	↕	R	R
	TSTB													5D	2	1	B − 00	•	•	↕	↕	R	R

Figure 3.30 (*continued overleaf*)

INDEX REGISTER AND STACK POINTER OPERATIONS	MNEMONIC	IMMED			DIRECT			INDEX			EXTND			INHER			BOOLEAN/ARITHMETIC OPERATION	5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Compare Index Reg	CPX	8C	3	3	9C	4	2	AC	6	2	BC	5	3				$(X_H/X_L) - (M/M+1)$	•	•	⑦	↕	⑧	•
Decrement Index Reg	DEX													09	4	1	$X - 1 \rightarrow X$	•	•	•	↕	•	•
Decrement Stack Pntr	DES													34	4	1	$SP - 1 \rightarrow SP$	•	•	•	•	•	•
Increment Index Reg	INX													08	4	1	$X + 1 \rightarrow X$	•	•	•	↕	•	•
Increment Stack Pntr	INS													31	4	1	$SP + 1 \rightarrow SP$	•	•	•	•	•	•
Load Index Reg	LDX	CE	3	3	DE	4	2	EE	6	2	FE	5	3				$M \rightarrow X_H, (M+1) \rightarrow X_L$	•	•	⑨	↕	R	•
Load Stack Pntr	LDS	8E	3	3	9E	4	2	AE	6	2	BE	5	3				$M \rightarrow SP_H, (M+1) \rightarrow SP_L$	•	•	⑨	↕	R	•
Store Index Reg	STX				DF	5	2	EF	7	2	FF	6	3				$X_H \rightarrow M, X_L \rightarrow (M+1)$	•	•	⑨	↕	R	•
Store Stack Pntr	STS				9F	5	2	AF	7	2	BF	6	3				$SP_H \rightarrow M, SP_L \rightarrow (M+1)$	•	•	⑨	↕	R	•
Indx Reg → Stack Pntr	TXS													35	4	1	$X - 1 \rightarrow SP$	•	•	•	•	•	•
Stack Pntr → Indx Reg	TSX													30	4	1	$SP + 1 \rightarrow X$	•	•	•	•	•	•

JUMP AND BRANCH OPERATIONS	MNEMONIC	RELATIVE			INDEX			EXTND			INHER			BRANCH TEST	5	4	3	2	1	0
		OP	~	#	OP	~	#	OP	~	#	OP	~	#		H	I	N	Z	V	C
Branch Always	BRA	20	4	2										None	•	•	•	•	•	•
Branch If Carry Clear	BCC	24	4	2										C = 0	•	•	•	•	•	•
Branch If Carry Set	BCS	25	4	2										C = 1	•	•	•	•	•	•
Branch If = Zero	BEQ	27	4	2										Z = 1	•	•	•	•	•	•
Branch If ≥ Zero	BGE	2C	4	2										$N \oplus V = 0$	•	•	•	•	•	•
Branch If > Zero	BGT	2E	4	2										$Z + (N \oplus V) = 0$	•	•	•	•	•	•
Branch If Higher	BHI	22	4	2										$C + Z = 0$	•	•	•	•	•	•
Branch If ≤ Zero	BLE	2F	4	2										$Z + (N \oplus V) = 1$	•	•	•	•	•	•
Branch If Lower Or Same	BLS	23	4	2										$C + Z = 1$	•	•	•	•	•	•
Branch If < Zero	BLT	2D	4	2										$N \oplus V = 1$	•	•	•	•	•	•
Branch If Minus	BMI	2B	4	2										N = 1	•	•	•	•	•	•
Branch If Not Equal Zero	BNE	26	4	2										Z = 0	•	•	•	•	•	•
Branch If Overflow Clear	BVC	28	4	2										V = 0	•	•	•	•	•	•
Branch If Overflow Set	BVS	29	4	2										V = 1	•	•	•	•	•	•
Branch If Plus	BPL	2A	4	2										N = 0	•	•	•	•	•	•
Branch To Subroutine	BSR	8D	8	2											•	•	•	•	•	•
Jump	JMP				6E	4	2	7E	3	3				See Special Operations	•	•	•	•	•	•
Jump To Subroutine	JSR				AD	8	2	BD	9	3					•	•	•	•	•	•
No Operation	NOP										01	2	1	Advances Prog. Cntr. Only	•	•	•	•	•	•
Return From Interrupt	RTI										3B	10	1		—	—	—	—	⑩	—
Return From Subroutine	RTS										39	5	1	See special Operations	•	•	•	•	•	•
Software Interrupt	SWI										3F	12	1		•	S	•	•	•	•
Wait for Interrupt	WAI										3E	9	1		•	⑪	•	•	•	•

CONDITIONS CODE REGISTER OPERATIONS	MNEMONIC	INHER			BOOLEAN OPERATION	5	4	3	2	1	0
		OP	~	#		H	I	N	Z	V	C
Clear Carry	CLC	0C	2	1	$0 \rightarrow C$	•	•	•	•	•	R
Clear Interrupt Mask	CLI	0E	2	1	$0 \rightarrow I$	•	R	•	•	•	•
Clear Overflow	CLV	0A	2	1	$0 \rightarrow V$	•	•	•	•	R	•
Set Carry	SEC	0D	2	1	$1 \rightarrow C$	•	•	•	•	•	S
Set Interrupt Mask	SEI	0F	2	1	$1 \rightarrow I$	•	S	•	•	•	•
Set Overflow	SEV	0B	2	1	$1 \rightarrow V$	•	•	•	•	S	•
Acmltr A → CCR	TAP	06	2	1	$A \rightarrow CCR$	—	—	—	⑫	—	—
CCR → Acmltr A	TPA	07	2	1	$CCR \rightarrow A$	•	•	•	•	•	•

LEGEND:

OP	Operation Code (Hexadecimal);	00	Byte = Zero;
~	Number of MPU Cycles;	H	Half-carry from bit 3;
#	Number of Program Bytes;	I	Interrupt mask
+	Arithmetic Plus;	N	Negative (sign bit)
−	Arithmetic Minus;	Z	Zero (byte)
•	Boolean AND;	V	Overflow, 2's complement
M_{SP}	Contents of memory location pointed to be Stack Pointer;	C	Carry from bit 7
		R	Reset Always
		S	Set Always
+	Boolean Inclusive OR;	↕	Test and set if true, cleared otherwise
⊕	Boolean Exclusive OR;	•	Not Affected
$\bar{M}$	Complement of M;	CCR	Condition Code Register
→	Transfer Into;	LS	Least Significant
0	Bit = Zero;	MS	Most Significant

CONDITION CODE REGISTER NOTES:

(Bit set if test is true and cleared otherwise)

① (Bit V) Test: Result = 10000000?

② (Bit C) Test: Result = 00000000?

③ (Bit C) Test: Decimal value of most significant BCD Character greater than nine? (Not cleared if previously set.)

④ (Bit V) Test: Operand = 10000000 prior to execution?

⑤ (Bit V) Test: Operand = 01111111 prior to execution?

⑥ (Bit V) Test: Set equal to result of N ⊕ C after shift has occurred.

⑦ (Bit N) Test: Sign bit of most significant (MS) byte of result = 1?

⑧ (Bit V) Test: 2's complement overflow from subtraction of LS bytes?

⑨ (Bit N) Test: Result less than zero? (Bit 15 = 1)

⑩ (All) Load Condition Code Register from Stack. (See Special Operations)

⑪ (Bit I) Set when interrupt occurs. If previously set, a Non-Maskable Interrupt is required to exit the wait state.

⑫ (ALL) Set according to the contents of Accumulator A.

Figure 3.30 (cont'd)

Problem 2 *Read and Print n characters*

Given a microprocessor, a printer and a paper tape reader design a system to allow n characters on the tape to be printed under program control.

Use the microprocessor wait/go mode to implement your design, which is to be verified using the INTEL 8080 and the MOTOROLA 6800.

8080 SOLUTION

Step 1 *Aim of the design*

To read and print n characters.

Step 2 *Device characteristics*

The microprocessor has wait/go logic. The reader and printer are action/status devices.†

Step 3 *System design*

The block diagram of our solution is shown in Figure 3.32. As the printer is an acceptor no I/O port is needed. The step-by-step operation of the system is flow-charted in Figure 3.33.

Step 4 *Hardware design*

With the exception of the I/O port for the reader in Figure 3.34 no other hardware is needed, since in the case the INTEL 8080 $e = w$. See example 1 in section 3.4.

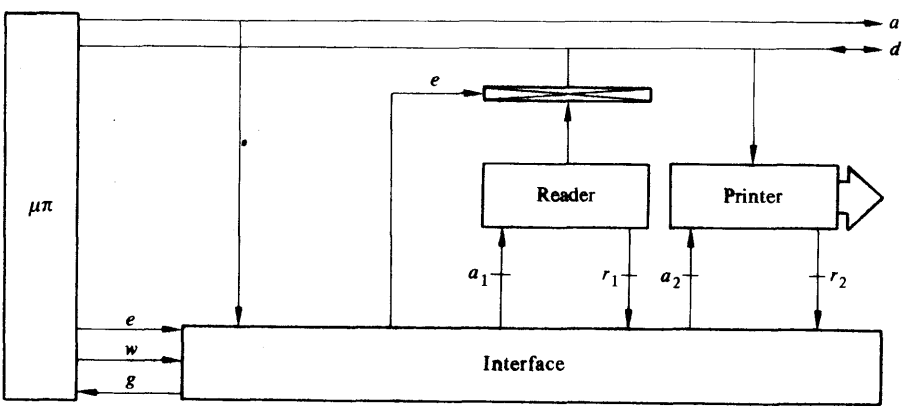


Figure 3.32.

†Action/status devices are explained in Appendix 1.

Step 5 *Software design*

The octal and hexadecimal listings for the INTEL 8080 are derived by direct reference to the flow chart in Figure 3.33 and to the programming chart in Figure 3.27 (or to the instruction set in Figure 3.28).

Octal address		*Octal listing*	*Hex listing*	*Mnemonics*	*Comments*
H	*L*				
003	000	016	*OE*	*MVI C*	Load register *C* with *n*.
	001	(*n*)	(*n*)		
	002	014	*OC*	*INR C*	Increment register *C*. Sets condition flags.
L2:	003	015	*OD*	*DCR C*	Decrement register *C*.
	004	312	*CA*	*JZ*	If register *C* is empty, jump to *L1*.
	005	016	*OE*	*L1*	
	006	003	*O3*		
	007	333	*DB*	*IN*	Read next character.
	010	010	*O8*		
	001	323	*D3*	*OUT*	Print character.
	012	020	18		
	013	303	*C3*	*JMP*	Jump to *L2*.
	014	003	03	*L2*	
	015	003	03		
L1:	016	166	76	*HLT*	Halt.

6800 SOLUTION

Step 1
Step 2
Step 3 — Same as in the 8080 solution

Step 4 *Hardware design*

The block diagram of our solution using the M6800 is shown in Figure 3.35. As in the previous problem, in addition to the I/O port we require an AND gate to generate the enable signal *e*, as explained in section 3.4—see circuits 1 and 2 of example 2 in that section.

Step 5 *Software design*

By reference to the flow chart in Figure 3.36 and to the programming chart in Figure 3.30 (or to the instruction set in Figure 3.31), we obtain the hexadecimal listing of our program. It is shown on page 86.

Problem 3 *Print a record*

Given a paper tape reader, a printer and a microprocessor, design a system that prints a record of *n* characters. The record label is 4–5–6. Character sequence 4–5–6 is used as a label only.

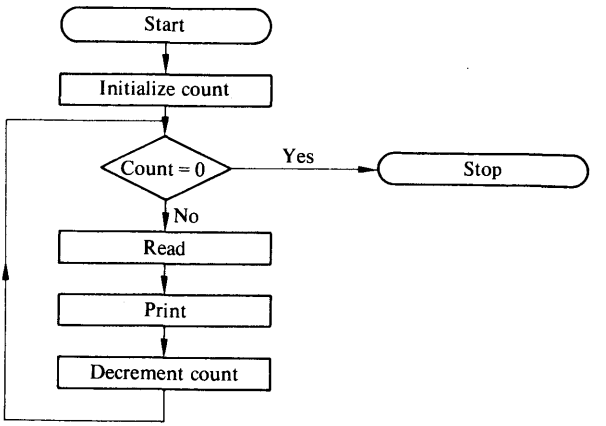


Figure 3.33.

Use the wait/go mode to implement your design, which is to be verified using the INTEL 8080 and the MOTOROLA 6800.

8080 SOLUTION

Step 1 *Aim of the design*

The aim of the design is to print a record.

Step 2 *Device characteristics*

The microprocessor has wait/go logic. The reader and printer are action/status devices.†

Step 3 *System design*

The block diagram of our solution is shown in Figure 3.32. Its step-by-step operation is flow-charted in Figure 3.36.

Step 4 *Hardware design*

With the exception of the I/O port in Figure 3.34 no other hardware is needed, since for the INTEL 8080 $e = w$. See example 1 in section 3.4.

Step 5 *Software design*

The octal and hexadecimal listings for the INTEL 8080 are derived by direct reference to the flow chart in Figure 3.36 and to the programming chart in Figure 3.27 (or to the instruction set in Figure 3.28).

†Action/status devices are discussed in Appendix 1.

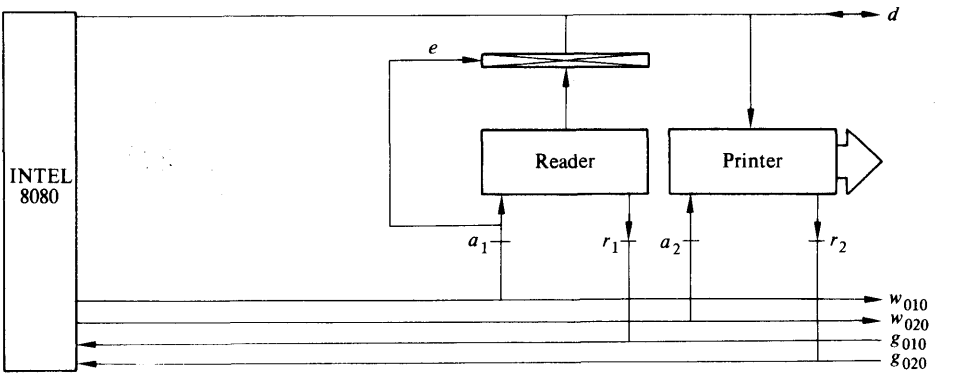


Figure 3.34.

	Hex address		*Hex listing*	*Mnemonics*	*Comments*
	H	*L*			
	00	00	*C6*	*LDA B,n*	Load register *B* with *n*.
		01	*n*	*n*	
L2:		02	27	*BEQ*	If register *B* = 0, jump to *L1*.
		03	*0B*	*L1*	
		04	*B6*	*LDA A,2000*	Read a character.
		05	20		
		06	00		
		07	01	*NOP*	
		08	*B7*	*STA A,4000*	Print the character.
		09	40		
		0*A*	00		
		0*B*	01	*NOP*	
		0*C*	5*A*	*DEC B*	Decrement *B*.
		0*D*	7*E*	*JMP L2*	Jump to *L2*.
		0*E*	00		
		0*F*	02		
L1:		10	3*F*	*SWI*	Stop.

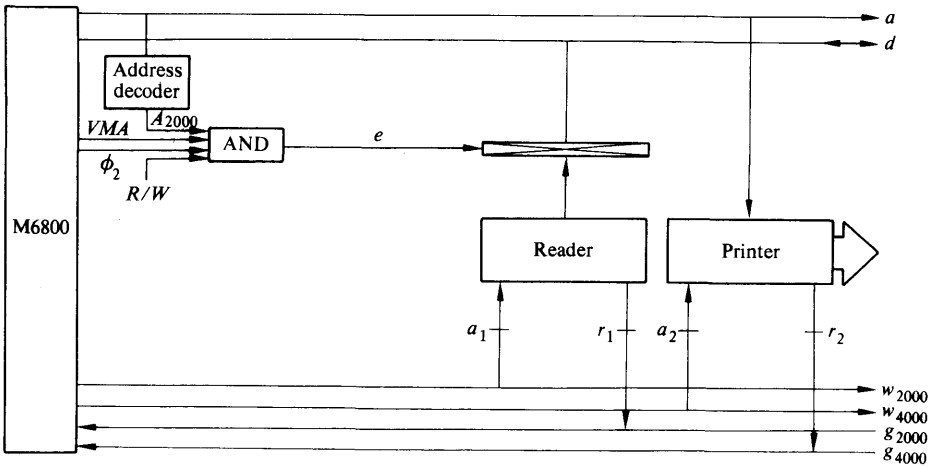


Figure 3.35.

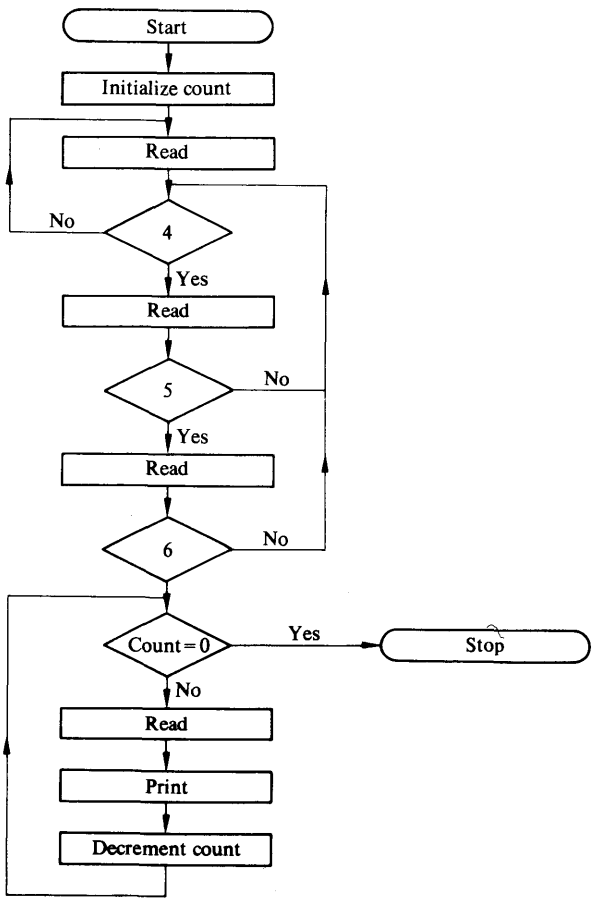


Figure 3.36.

	Octal address		*Octal listing*	*Hex listing*	*Mnemonics*	*Comments*
	H	*L*				
L1:	003	000	333	*DB*	*IN*	Read next character.
		001	020	10		
L2:		002	376	*FE*	*CPI*	Compare *AC* with 4.†
		003	264	*B4*		
		004	302	*C2*	*JNZ*	If character not a 4, jump to *L1*.
		005	000	00	*L1*	
		006	003	03		
		007	333	*DB*	*IN*	Read next character.
		010	020	10		
		011	376	*FE*	*CPI*	Compare *AC* with 5.†
		012	265	*B5*		
		013	302	*C2*	*JNZ*	If character not a 5, jump to *L2*.
		014	002	02	*L2*	
		015	003	03		
		016	333	*DB*	*IN*	Read next character.
		017	020	10		
		020	376	*FE*	*CPI*	Compare *AC* with 6.†
		021	266	*B6*		
		022	302	*C2*	*JNZ*	If character not a 6, jump to *L2*.
		023	002	02	*L2*	
		024	003	03		
		025	016	*OE*	*MVI C*	Load register *C* with *n*.
		026	(*n*)	(*n*)	*n*	
		027	014	*OC*	*INR C*	Increment register *C*. Sets condition flags.
L4:		030	015	*OD*	*DCR C*	Decrement register *C*.
		031	312	*CA*	*JZ*	If register *C* is empty, jump to *L3*.
		032	043	23	*L3*	
		033	003	03		
		034	333	*DB*	*IN*	Read next character.
		035	020	10		
		036	323	*D3*	*OUT*	Print character.
		037	030	18		
		040	303	*C3*	*JMP*	Jump to *L4*.
		041	030	18	*L4*	
		042	003	03		
L3:		043	166	76	*HLT*	Halt.

†See ASCII Table on p. 74.

6800 SOLUTION

Step 1
Step 2
Step 3
} Same as in the 8080 solution

Step 4 *Hardware design*

The block diagram of our solution using the M6800 is shown in Figure 3.35. As in the previous problem, in addition to the I/O port we require an AND gate to generate the enable signal *e*, as explained in section 3.4—see circuits 1 and 2 of example 2 in that section.

Step 5 *Software design*

By reference to the flow chart in Figure 3.36 and to the programming chart in Figure 3.30 (or to the instruction set in Figure 3.31), we obtain the hexadecimal listing of our program. It is shown below.

Hex address			*Hex listing*	*Mnemonics*	*Comments*
	H	*L*			
L1:	00	00	*B6*	*LDA A,2000*	Read next character.
		01	20		
		02	00		
		03	01	*NOP*	
L2:		04	81	*CMP A*	Compare *A* with 4.
		05	34		
		06	26	*BNE L1*	If character not a 4, jump to *L1*.
		07	*F8*		
		08	*B6*	*LDA A,2000*	Read next character.
		09	20		
		0*A*	00		
		0*B*	01	*NOP*	
		0*C*	81	*CMP A*	Compare *A* with 5.
		0*D*	35		
		0*E*	26	*BNE L2*	If character not a 5, jump to *L2*.
		0*F*	*F4*		
		10	*B6*	*LDA A,2000*	Read next character.
		11	20		
		12	00		
		13	01	*NOP*	
		14	81	*CMP A*	Compare *A* with 6.
		15	36		
		16	26	*BNE L2*	If character not a 6, jump to *L2*.
		17	*EC*		
		18	*C6*	*LDA B,n*	Load register *B* with *n*.
		19	*n*	*n*	
L3:		1A	27	*BEQ*	If register *B* = 0, jump to *L4*.
		1B	0B	*L4*	
		1C	*B6*	*LDA A,2000*	Read a character.
		1D	20		
		1E	00		
		1F	01	*NOP*	

Hex address		*Hex listing*	*Mnemonics*	*Comments*
H	*L*			
	20	*B7*	*STA A*,4000	Print the character.
	21	40		
	22	00		
	23	01	*NOP*	
	24	5*A*	*DEC B*	Decrement *B*.
	25	7*E*	*JMP L*3	Jump to *L*3.
	26	00		
	27	1*A*		
*L*4:	28	3*F*	*SWI*	Stop.

3.6 REFERENCES

1. 'M6800 Microprocessor Applications Manual,' Motorola, 1975.
2. Zissos, D. and Duncan, F. G. 'Microprocessor Interfaces,' *Electronics Letters*, vol. **12**, No. 23, November 11, 1976.
3. Zissos, D. 'Problems and Solutions in Logic Design', Oxford University Press, 1976.
4. *INTEL* 8080 Microprocessor Systems User's Manual, September 1975.
5. Zissos, D., Bathory, J. C. 'Wait/go Microprocessor Systems,' *Proceedings Mimi* 1977, November 1977.

4

Test-and-Skip Systems

In this chapter we outline step-by-step methods for the design and implementation of test-and-skip microprocessor systems. The clock stretching method for achieving I/O synchronization is discussed. The design philosophy adopted is outlined in Chapter 2 (section 2.9) and the design steps we use are described in section 2.10 of the same chapter.

4.1 INTRODUCTION

The need to synchronize the microprocessor with a peripheral during an I/O operation has been explained in Chapters 2 and 3. One method of implementing I/O synchronization, the wait/go method, was described in the previous chapter. In this chapter we shall describe the two alternative methods, namely *test-and-skip* and *clock stretching.*

In the test-and-skip mode we synchronize the microprocessor operation with the response of a peripheral using a *software loop.* It works as follows. After each I/O instruction the programmer inputs the status of the peripheral and tests whether it has fully responded or not. If not, the programmer repeats the test. He continues to do so until the device becomes ready, at which point the program comes out of the loop and proceeds to execute the next instruction, as shown in Figure 4.1. The design and implementation of microprocessor systems using this mode of operation is described in the next section.

In a limited number of cases where I/O synchronization can be achieved by slowing down the microprocessor clock frequency, a method commonly referred to as *clock stretching* can be used. We explain this method in section 4.3.

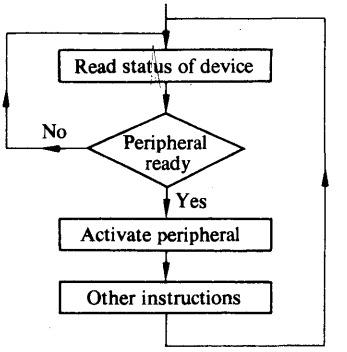


Figure 4.1.

4.2 TEST-AND-SKIP SYSTEMS

The block diagram of a test-and-skip system is shown in Figure 4.2. Signal r indicates the availability/non-availability of the peripheral; $r = 1$ when the peripheral is ready, otherwise $r = 0$. Its step-by-step operation is as follows. The programmer executes an I/O instruction which activates the peripheral, causing signal r to change from logic 1 to logic 0. He next inputs the status word, one bit of which is signal r. He then tests to check whether r equals 1 or not. If r does not equal 1, indicating that the peripheral has not fully responded

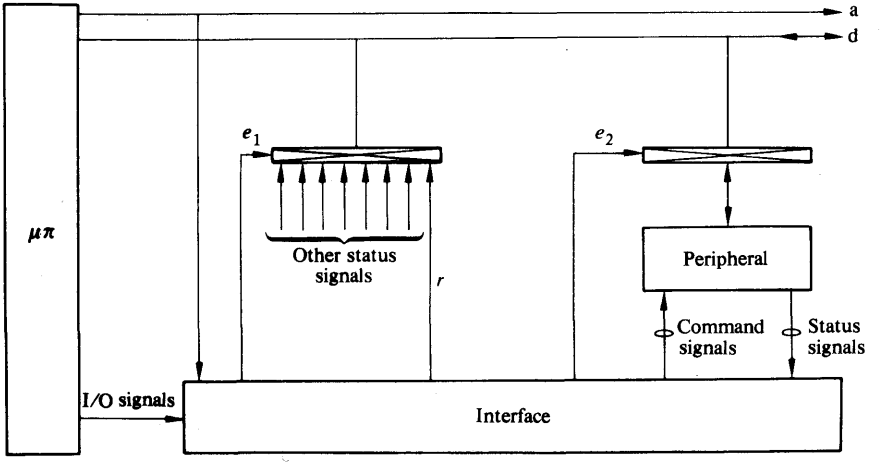


Figure 4.2.

yet, he inputs the status word again and repeats the test. He continues to do so until r equals 1, at which point the program is resumed as shown in Figure 4.1. Note that the duration of the test-and-skip loop equals the response time of the peripheral.

4.3 CLOCK STRETCHING

Let us denote by f_{max} and f_{min} the maximum and minimum clock frequencies in Hz within which the microprocessor can operate. We use variable f to denote the operating frequency, that is $f_{min} \leq f \leq f_{max}$.

Now if the response time of the device is t seconds, where

$$\frac{1}{f_{max}} \leq t \leq \frac{1}{f_{min}},$$

we can clearly synchronize the microprocessor with the device by changing its clock frequency to f, where

$$\frac{1}{f} \geq t.$$

If we assume that the microprocessor interstate transitions occur on the trailing edge of clock signal ϕ_1, then we use ϕ_2 as our clock in our clock-stretching circuit. The implementation of such a circuit is straightforward and

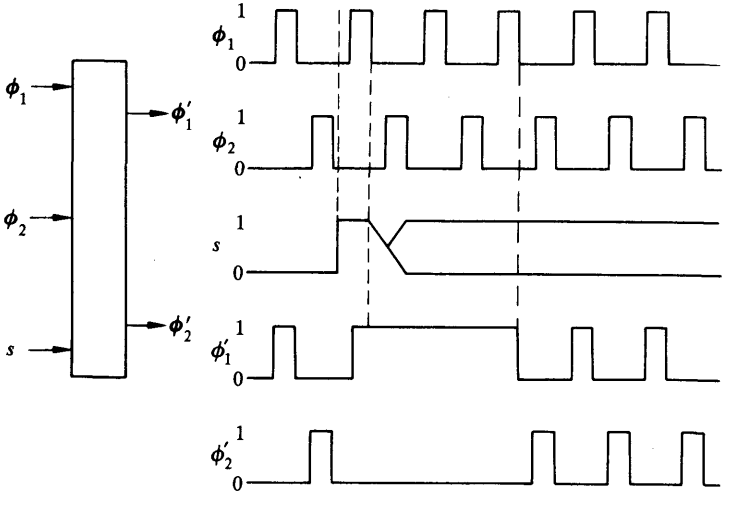


Figure 4.3.

should present no difficulty to the reader who possesses a working knowledge of logic design. The method is described in Chapter 1 of this book and in more detail in Chapter 4 of 'Problems and Solutions in Logic Design' by D. Zissos, Oxford University Press, 1976.

To illustrate the simplicity of the method, we shall show how we can stretch the microprocessor clock frequency by two clock cycles. If we denote by variable s the signal generated by the interface to request a 'two-cycle stretch', the corresponding I/O characteristics are shown in Figure 4.3. In Figure 4.4 we show the internal-state diagram. By direct reference to the internal-state diagram, we obtain

$$S_A = S_1 = \bar{A}\cdot B, \qquad \text{therefore } J_A = B$$

$$R_A = S_3\cdot\bar{s} = A\cdot\bar{B}\cdot\bar{s}, \qquad \text{therefore } K_A = \bar{B}\cdot\bar{s}$$

$$S_B = S_0\cdot s = \bar{A}\cdot\bar{B}\cdot s, \qquad \text{therefore } J_B = \bar{A}\cdot s$$

$$R_B = S_2 = A\cdot B, \qquad \text{therefore } K_B = A$$

$$\begin{aligned}\phi_1' &= S_0\cdot\phi_1 + S_1 + S_2 + S_3\cdot\phi_1\\ &= \bar{A}\cdot\bar{B}\cdot\phi_1 + \bar{A}\cdot B + A\cdot B + A\cdot\bar{B}\cdot\phi_1\\ &= B + \phi_1\end{aligned}$$

$$\begin{aligned}\phi_2' &= (S_0 + S_3)\cdot\phi_2\\ &= (\bar{A}\cdot\bar{B} + A\cdot\bar{B})\cdot\phi_2\\ &= \bar{B}\cdot\phi_2\end{aligned}$$

The corresponding circuit is shown in Figure 4.5.

4.4 PROBLEMS AND SOLUTIONS

In this section we demonstrate our design steps by means of problems and fully-worked out solutions. The reader's attention is drawn to the fact that,

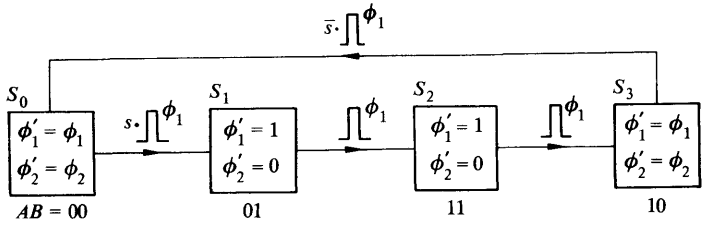


Figure 4.4.

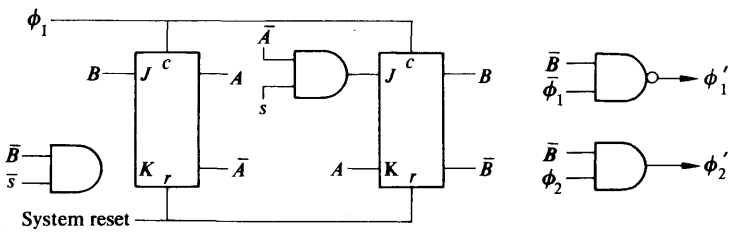


Figure 4.5.

although we use the INTEL 8080 to implement our designs, the procedures used apply to all types of microprocessors. Specifically it should be noted that the first three steps in the design are executed without reference to the microprocessor.

Problem 1 *RAM to printer*

Design an interface to allow a programmer to transfer data from consecutive locations in RAM or ROM through the m.p.u. onto an acceptor.

The acceptor in our case is a digital printer, whose terminal characteristics are described in Figure 4.6. The microprocessor is the INTEL 8080.

SOLUTION

Step 1 *Aim of the design*

The aim is to design and implement an interface between a microprocessor and a relatively simple peripheral using the test-and-skip mode of microprocessor operation.

Step 2 *Device characteristics*

The relevant I/O signals of the INTEL 8080 are shown in Figure 4.7. The terminal characteristics of the printer are shown in Figure 4.6.

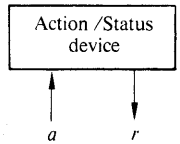
Step 3 *System design*

The block diagram of our solution is shown in Figure 4.8. Its operation is flow-charted in Figure 4.9.

Step 4. *Hardware design*

The hardware consists of an address decoder, a NAND gate, an inverter and an AND gate as shown in Figure 4.10. The decoder is used to decode the I/O address of the printer and the tristate. They can both be allocated the same address because one is an input device and the other is an output device. If we

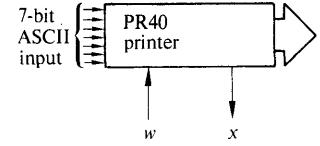
Action/status devices



Terminal a. A 0 to 1 transition on this line activates device.

Terminal r. Status signal r is 1 when the device is ready, otherwise $r = 0$. Activation of the device is not possible when $r = 0$.

Printer



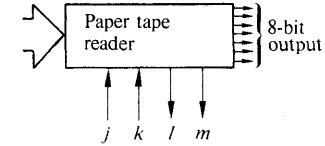
Terminal w. A negative-going pulse of one microsecond minimum duration loads the character into the buffer, or activates the printing mechanism if either the buffer is full or the ASCII code† for carriage return '015' in octal is being input.

Non-ASCII characters are ignored.

Terminal x. Status signal x equals 1 when the printer is ready and the *w* line is at 1.

†See Figure 3.23 (p. 74).

Reader



Terminal j. A ground on this terminal causes the tape to move left.

Terminal k. A ground on this terminal causes the tapes to move right.

Terminal l. Status signal l is 1 when sprocket hole is under the read head, otherwise $l = 0$.

Terminal m. A logic 1 indicates that the reader is available for the next drive pulse.

To stop tape on next character remove drive within 1 msec after the leading edge of signal *l*. Its minimum duration is 1 μsec.

Figure 4.6.

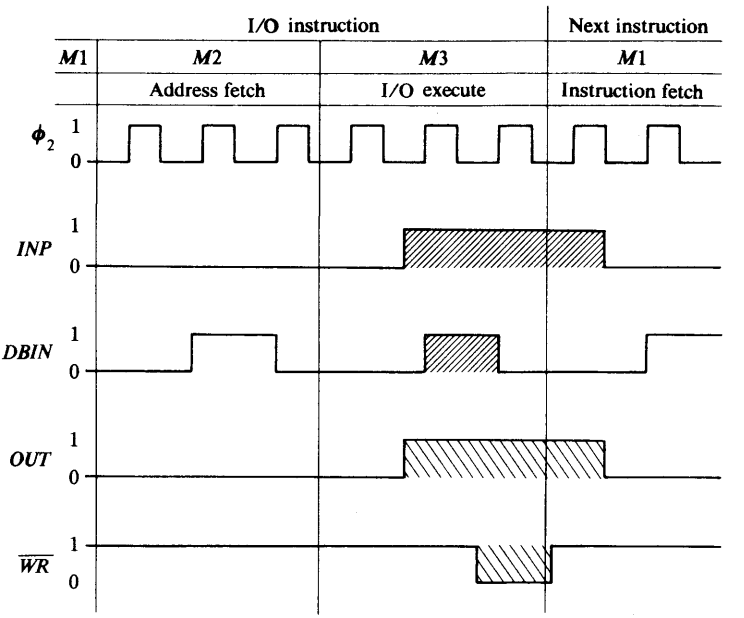


Figure 4.7.

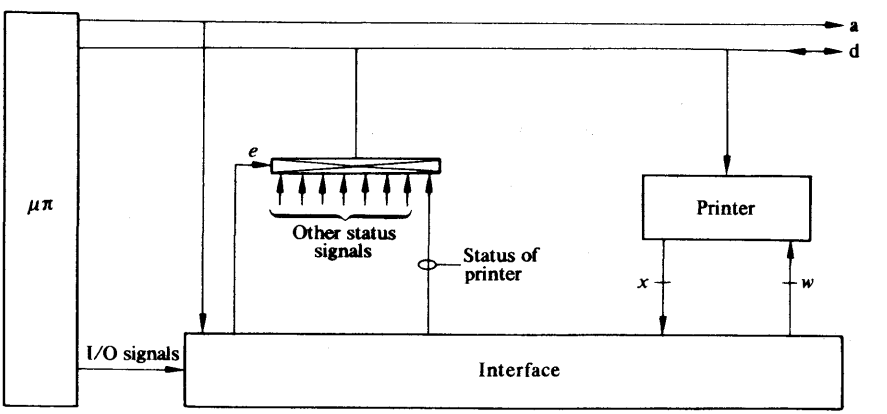


Figure 4.8.

denote the address by Am, then $w = \overline{Am} \cdot \overline{OUT} \cdot \overline{WR}$; this signal is implemented by the NAND gate. We open the tristate when I/O *IN* instruction with address Am is being executed i.e. $e = IN \cdot DBIN \cdot Am$. This signal is generated by the AND gate. Since the printer status signal x can only be monitored when $w = 1$, we make w equal to $\overline{OUT}$. This reduces the time when signal w cannot be monitored to the duration of the OUT pulse. In the arrangement we have used, the eight tristate outputs are all zero when the printer is unready.

Step 5 *Software design*

For the sake of clarity we shall first design the software required to synchronize the printer operation with the microprocessor. The flow chart is shown in Figure 4.1. The mnemonics are shown below.

	OUT	Transfer character from
	#	*AC* to printer.
*L*1:	*IN*	Input the status of
	#	the printer.
	ANA A	Set flags.
	JZ	If printer not
	*L*1	ready go to *L*1.

By direct reference to our flow chart in Figure 4.9 and either to the programming chart in Figure 3.27 or to the instruction set in Figure 3.28, we obtain the octal and hexadecimal listings of our program. These are shown below ($Am = A010$).

	Octal address		*Octal*	*Hex*	*Mnemonics*	*Comments*
	H	*L*				
	003	000	041	21	*LXI H*	Load immediate the initial RAM address into double register *HL*.
		001	(*L*)	(*L*)		
		002	(*H*)	(*H*)		
		003	016	*OE*	*MVI C*	Load block length into register *C*.
		004	(*n*)	(*n*)		
		005	014	*OC*	*INR C*	Increment register *C* to set condition flags.
*L*3:		006	015	*OD*	*DCR C*	Decrement register *C*.
		007	312	*CA*	*JZ*	If register *C* is empty, jump to location *L*1.
		010	027	17	*L*1	
		011	003	03		
*L*2:		012	333	*DB*	*IN*	Read status of printer.
		013	010	08		
		019	247	*A7*	*ANA A*	Set flags.
		015	312	*CA*	*JZ*	If printer not ready, jump to location *L*2.
		016	012	*OA*	*L*2	
		017	003	03		

	Octal address	*Octal*	*Hex*	*Mnemonics*	*Comments*
	020	176	7E	*MOV A,m*	Move next character from memory to *AC*.
	021	323	03	*OUT*	
	022	010	08		Transfer character to printer.
	023	054	2C	*INR L*	Increment RAM address.
	024	303	C3	*JMP*	Jump to location *L3*
	025	006	06	*L3*	
	026	003	03		
L1:	027	076	3E	*MVI A*	Move immediate into *AC* code for carriage return, 015_8.
	030	015	8D		
	031	323	D3	*OUT*	Print. The contents of the *AC* are printed and the carriage is returned.
	032	010	08		
	033	166	76	*HLT*	Halt.

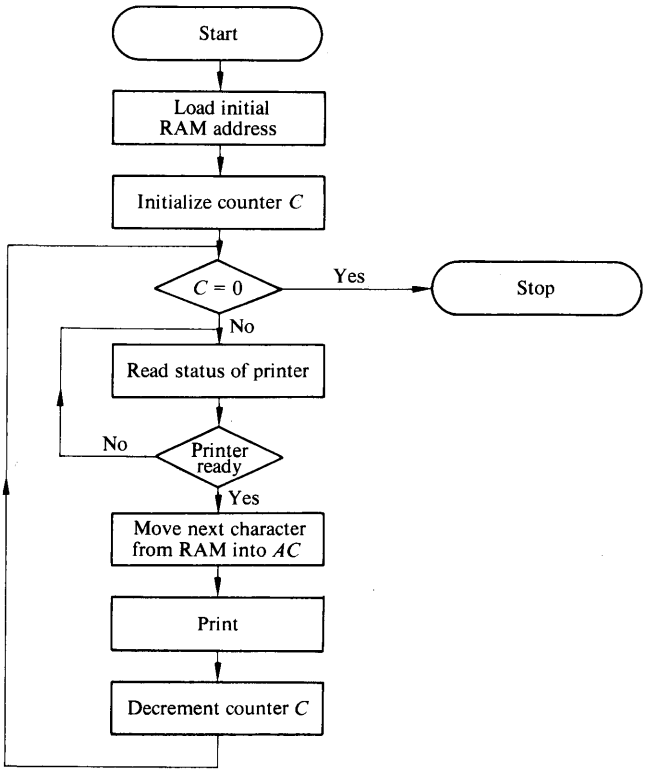


Figure 4.9.

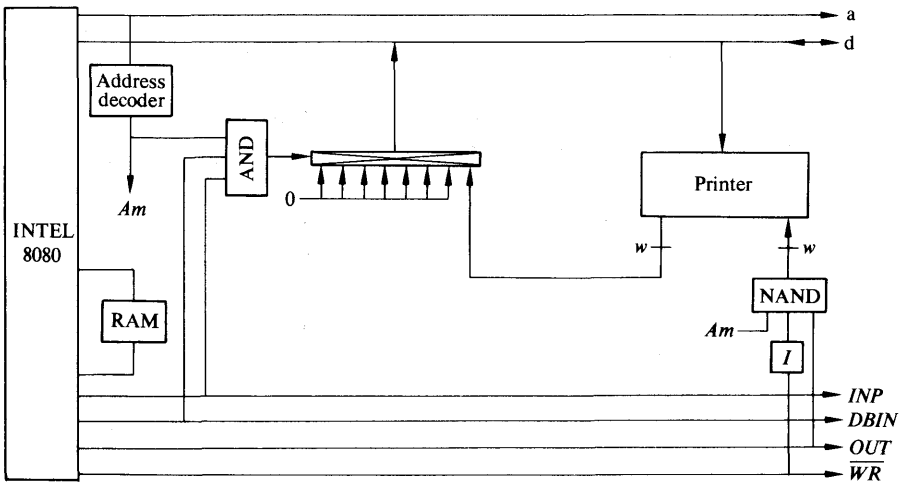


Figure 4.10.

Problem 2 *Reader to RAM*

Design an interface to allow a programmer to transfer data from a source through the m.p.u. into consecutive locations in RAM.

The data source in our case is a paper tape reader described in Figure 4.6, and the microprocessor is the INTEL 8080.

SOLUTION

Step 1 *Aim of the design*

The main aim is to design an interface between a microprocessor and a peripheral whose input signals have time restrictions.

Step 2 *Device characteristics*

The relevant I/O signals of the INTEL 8080 are shown in Figure 4.7. The reader's block diagram and terminal characteristics are shown in Figure 4.6.

Step 3 *System design*

The block diagram of our solution is shown in Figure 4.11. Its operation is flow-charted in Figure 4.12. The eight outputs of the status tristate are arranged to be all zero when the reader is not ready.

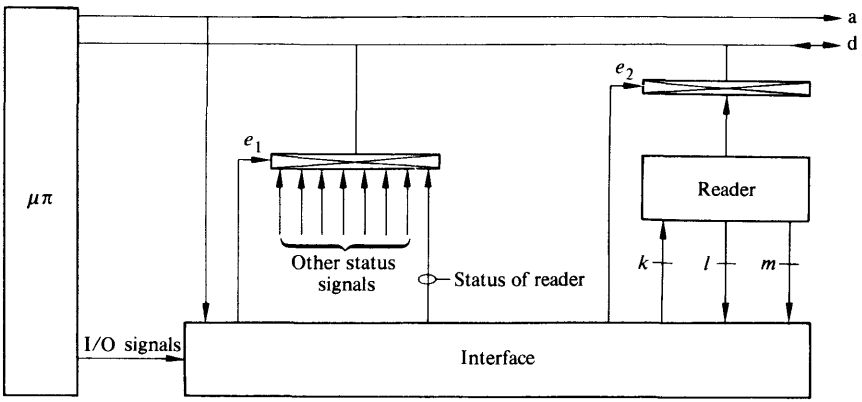


Figure 4.11.

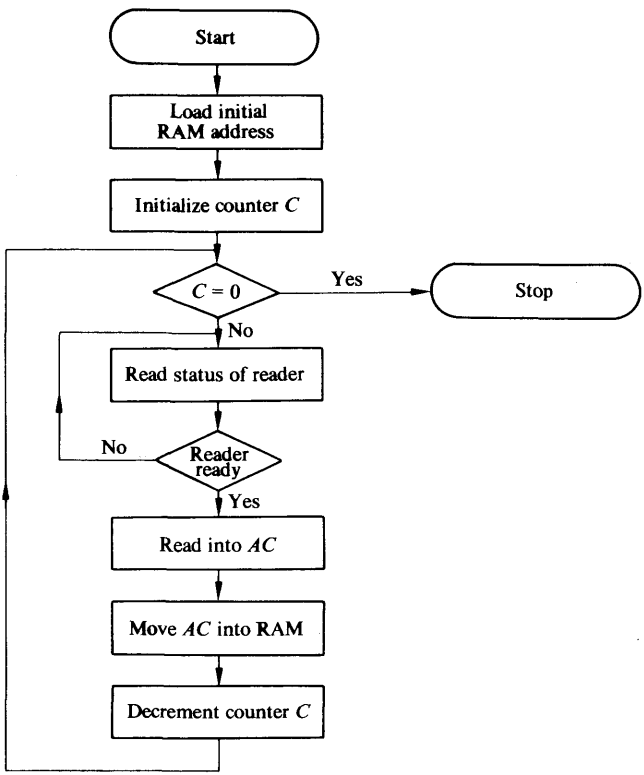


Figure 4.12.

Step 4 *Hardware design*

Because the minimum duration of an I/O pulse is 0.48 μsecs it cannot drive directly the reader. This time constraint can be overcome in practice by using the I/O pulse to set a flip-flop. Reference to the tape reader's terminal characteristics indicates that the drive signal can be removed when l changes to 0. Therefore we can reset the flip-flop with the 1 to 0 transition in signal l, as shown in Figure 4.13. The enable signals of the two tristates are

$$e_1 = INP \cdot DBIN \cdot A_q, \text{ and}$$

$$e_2 = INP \cdot DBIN \cdot A_p$$

They are implemented by the two AND gates in Figure 4.13.

Step 5 *Software design*

By direct reference to our programming flow chart in Figure 4.14 and either to the programming chart in Figure 3.27 or to the instruction set in Figure 3.28, we obtain the octal and hexadecimal listings of our program. These are shown below. ($A_p = A_{020}$ and $A_q = A_{030}$).

	Octal Address	*Octal*	*Hex.*	*Mnemonics*	*Comments*
	H L				
	003 000	041	21	*LXI H*	Load initial RAM address into *HL* register pair.
	001	(*L*)	(*L*)		
	002	(*H*)	(*H*)		
	003	016	*OE*	*MVI C*	Load block length into register *C*.
	004	(*n*)	(*n*)		
	005	014	*OC*	*INR C*	Increment register *C* to set condition flags.
L3:	006	015	*OD*	*DCR C*	Decrement register *C*.
	007	312	*CA*	*JZ*	If register *C* is empty, jump to location *L1*.
	010	027	17	*L1*	
	011	003	03		
L2:	012	333	*DB*	*IN*	Read status of reader.
	013	020	*IO*		
	014	247	*A7*	*ANA A*	Set flags.
	015	312	*CA*	*JZ*	If reader not ready, jump to location *L2*.
	016	012	*OA*	*L2*	
	017	003	03		
	020	333	*DB*	*IN*	Read next character.
	021	030	18		
	022	167	77	*MOV M,A*	Move *AC* contents to memory
	023	054	2*C*	*INX H*	Increment RAM address.
	024	303	*C3*	*JMP*	Jump to location *L3*.
	025	006	06	*L3*	
	026	003	03		
L1:	027	166	76	*HLT*	Halt.

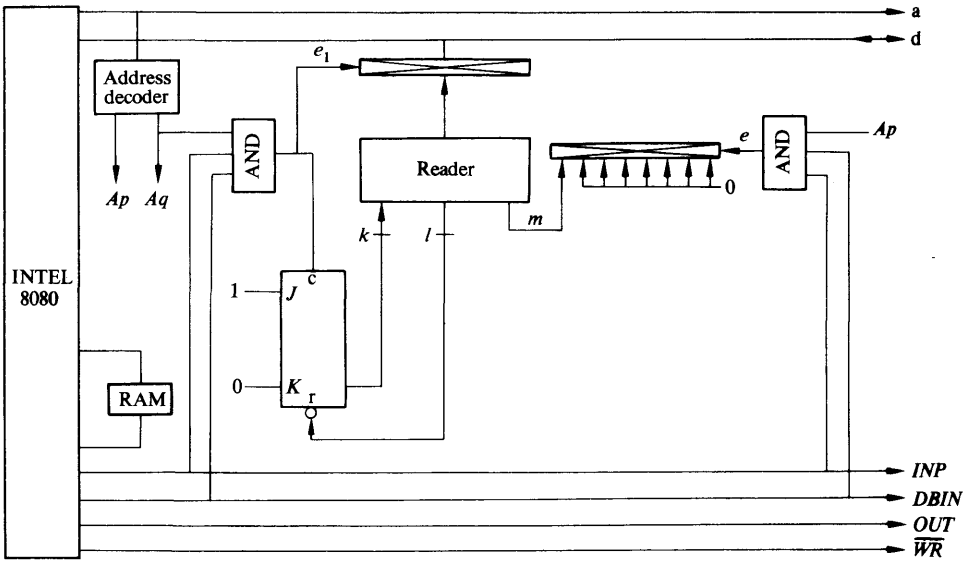


Figure 4.13.

Problem 3 *Read and print n-characters*

Design an interface between the paper tape reader used in problem 2 and the printer used in problem 1 to allow n characters to be printed—n is a variable defined by the programmer.

Use the INTEL 8080 to verify your design.

SOLUTION

Step 1 *Aim of the design*

The general aim is to design an interface between a microprocessor and two devices.

Step 2 *Device characteristics*

The relevant I/O signals of the INTEL 8080 are shown in Figure 4.7.

The terminal characteristics of the printer and reader are shown in Figure 4.6.

Step 3 *System design*

The block diagram of our solution is shown in Figure 4.15. The method we shall adopt consists of reading a character, printing it, decrementing a counter and waiting for both the reader and printer to respond fully at which time the

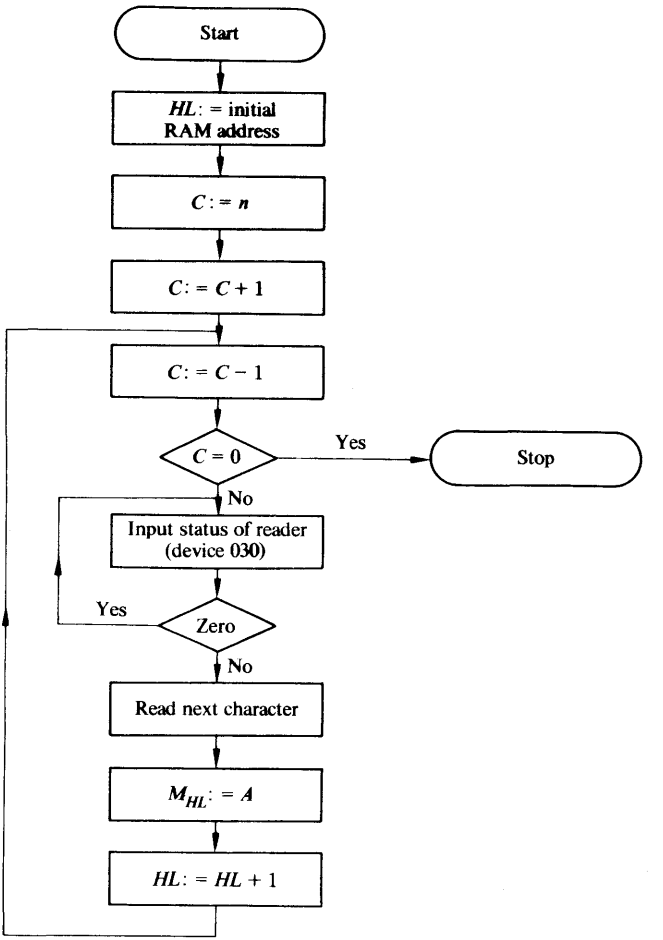


Figure 4.14.

process is repeated. When the character count becomes zero, indicating that n characters have been read and printed, we halt the operation, as shown in our flow chart in Figure 4.16.

Step 4 *Hardware design*

Reference to the reader's terminal characteristics reveals that to move the paper tape one character position we need to ground terminal k for at least 1 μ sec and must remove the ground before signal l changes to 1. As the duration of an I/O pulse is approximately 0.5 μsecs it cannot be used directly. The most

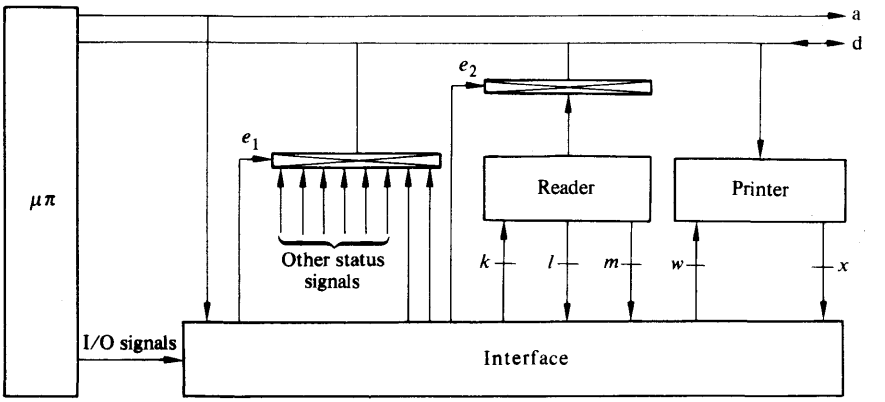


Figure 4.15.

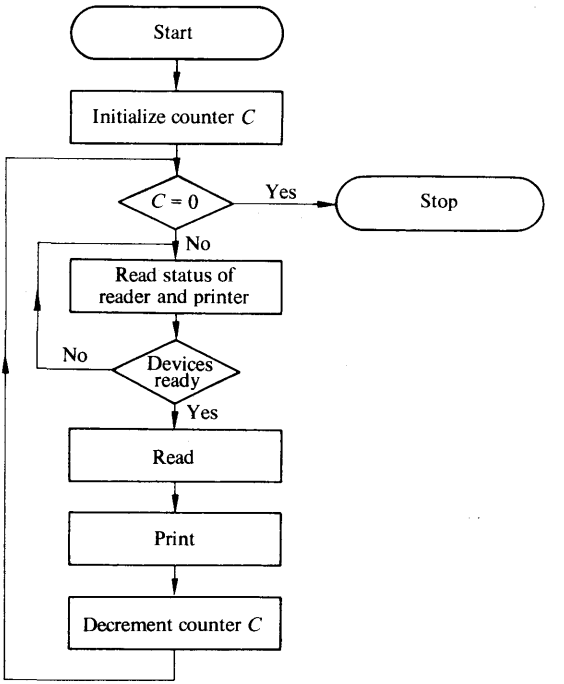


Figure 4.16.

straightforward method is to use the I/O pulse to set a flip-flop, and use the 1 to 0 transition in signal l to reset it. This transition occurs 2 msecs after k is grounded. This makes the duration of signal k about 2 msecs. If we allocate 003 as an I/O address to our reader, the I/O pulse $INP{\cdot}DBIN{\cdot}A_{003}$ can be used for this purpose. This sets the *JK* flip-flop during its 1 to 0 transition. This signal is used also to enable the reader tristate; that is $e_1 = INP{\cdot}DBIN{\cdot}A_{003}$.

To activate the printer we ground its terminal w when an I/O instruction with address 004 is executed, that is $w = \overline{OUT{\cdot}WR{\cdot}A004}$. This signal is implemented by the NAND gate and the inverter shown in Figure 4.17. Status signals m and x are fed into the second tristate. The output of tristate 2, when enabled, is '000 000 11' or '003_8', unless either or both of the devices (reader and printer) are unready. This gives the programmer the opportunity to enter a software wait loop until both the devices become ready.

In our solution we shall allocate the status port I/O address #005.

Step 5 *Software design*

The octal and hexadecimal listings for the INTEL 8080 derived by reference to the programming flow chart in Figure 4.18 and either to the instruction set in Figure 3.27 or to the programming chart in Figure 3.28, are

	Octal address		Octal	Hex	Mnemonics	Comments
	H	*L*	01			
	003	000	016	01	*MVI*	Move next byte into register *C*.
		001	(*n*)	(*n*)		*n* = number of characters.
		002	014	*OC*	*INR C*	Increment register *C* to set condition flags.
L3:		003	015	*OD*	*DCR C*	Decrement register *C*.
		004	312	*CA*	*JZ*	If register *C* is empty, jump to location *L1*.
		005	025	15	*L1*	
		006	003	03		
L2:		007	333	*DB*	*IN*	Input status.
		010	005	05		
		011	376	*FE*	*CPI*	Compare *AC* with next byte.
		012	003	03		
		013	302	*C2*	*JNZ*	If devices unready,
		014	007	07	*L2*	jump to location *L2*.
		015	003	03		
		016	333	*DB*	*IN*	Read a character.
		017	003	03		
		020	323	*D3*	*OUT*	Print the character.
		021	004	04		
		022	303	*C3*	*JMP*	Jump to
		023	003	03	*L3*	location *L3*.
		024	003	03		
L1:		025	166	76	*HLT*	Halt.

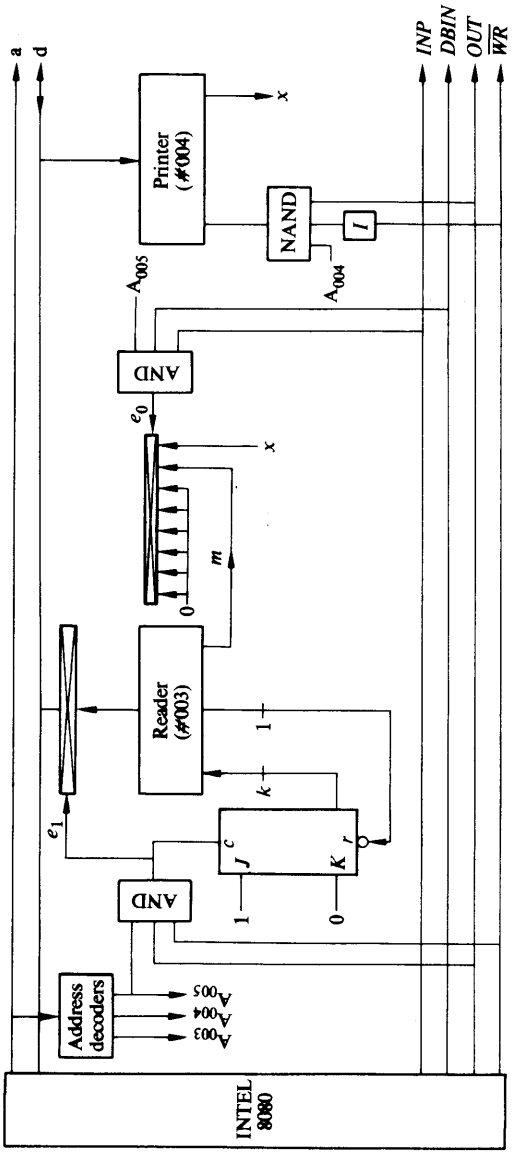

a
d
INP
DBIN
OUT
WR
Printer (#004)
x
NAND
A004
A005
AND
e0
x
m
0
Reader (#003)
1
k
e1
J
c
K
r
AND
1
0
Address decoders
A003
A004
A005
INTEL 8080

Figure 4.17.

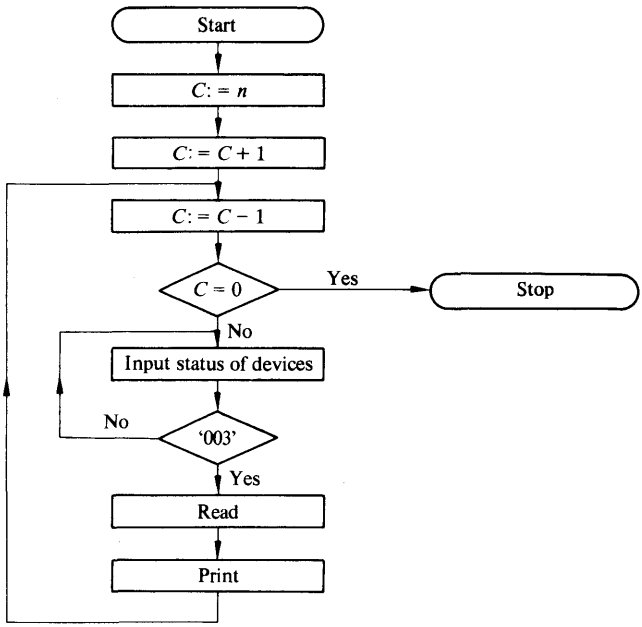

Start
C: = n
C: = C + 1
C: = C - 1
C = 0
Yes
Stop
No
Input status of devices
No
'003'
Yes
Read
Print

Figure 4.18.

5

Interrupt Systems

In this chapter we outline step-by-step methods for the design and implementation of interrupt systems. The design philosophy adopted and the design steps are as outlined in Chapter 2, sections 2.9 and 2.10, respectively.

5.1 INTRODUCTION

In this mode of operation an external event signals the microprocessor that it wishes it to suspend execution of its current program and to execute instead a different set of instructions, *the interrupt routine*, as shown in Figure 5.1. When the interrupt request is serviced, the interrupted program is resumed. For example a fire detector may signal the microprocessor that it has detected a fire. In such a case the microprocessor would suspend execution of its current program and proceed to execute a service routine that would typically trip alarm bells, warn personnel in the vicinity, turn on sprinklers, alert the fire brigade and so on. After the microprocessor has responded to the fire alarm, it returns to the interrupted program.

For reference purposes we denote by A_m the location in memory where the last instruction in the interrupted program resides, and by A_s the location of the first instruction in the interrupt routine. Note that at the point of interruption the contents of the program counter, *PC*, are A_{m+1}. This is because *PC* is incremented during machine cycle *M*1, as can be seen in Figure 2.4.

It follows that to switch from the main program to the interrupt routine, we simply replace the contents of the program counter, *PC*, by A_s. Similarly, we return to the interrupted program at the end of the interrupt routine by loading the program counter, *PC*, with A_{m+1}. This is the minimum information required by the microprocessor to resume the interrupted program; we shall

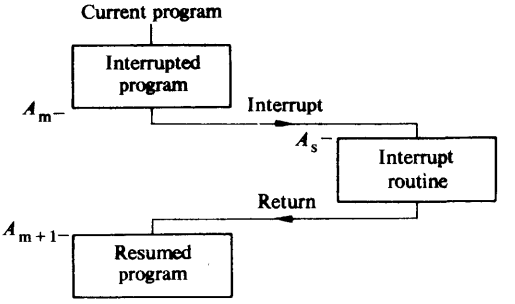


Figure 5.1.

refer to it as the program's *re-entry point*. In practice, the status of the condition flags and of the working registers must also be preserved during an interrupt routine. *Working registers* are m.p.u. registers that are used by both the interrupted program and the interrupt routine. The contents of the program counter, the status of the condition flags, and the status of the working registers we shall refer to collectively as *the program's re-entry point.*[1]

The primary reasons for interrupting a program in practice are to initiate, service, or terminate some process which is capable of being carried out simultaneously (in parallel) with the execution of a program.

Although the design of interrupt systems is carried out in well-defined steps, their implementation requires relatively more complicated hardware and software than any other microprocessor mode.

5.2 INTERRUPT SYSTEMS

The block diagram of our interrupt system is shown in Figure 5.2. It consists of an interrupt circuit,† shared by all the devices using the interrupt mode, a peripheral and its interrupt interface. It has been designed as a general system to accommodate any type of microprocessor and any type of peripheral. Its step-by-step operation is flow-charted in Figure 5.3 and summarized below.

When a peripheral requires servicing, or it is ready to transfer data in or out of the microprocessor, its interface, which monitors its signals, raises a *flag*. Flags are defined in the next section. The interrupt circuit, which monitors all the flags, then generates *the interrupt signal*, signal I in Figure 5.2, and some meaningful information, which we denote by variable i. Signal I informs the microprocessor that one or more peripherals wish to communicate with it. If

† We shall refer to it as interrupt logic.

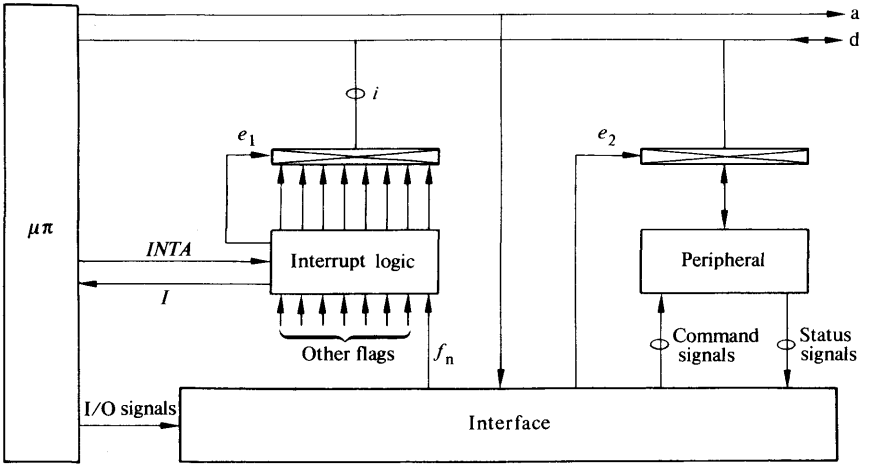


Figure 5.2.

interrupts are enabled, the microprocessor completes its current instruction and responds in the following manner.

1. It generates a signal to indicate that the program has been interrupted. We refer to this signal as *Interrupt Acknowledge*, and denote it by *INTA*, as shown in Figure 5.2.
2. Further interrupts are automatically disabled. This ensures that the microprocessor will not be interrupted again until the programmer is ready to accept another interrupt.
3. The re-entry point is stored on stack.
4. The source of interruption is identified by inputting *i*.
5. Working registers are stored on stack.
6. The request is serviced.
7. The interrupt flag is cleared.
8. The working registers are restored.
9. Interrupts are enabled.
10. The interrupted program is resumed (*PC* loaded with re-entry point).

Although the above sequence of events is typical, variations in the implementation of the individual steps exist depending on which microprocessor is being used. For example, in the Motorola 6800 all the m.p.u. registers and condition flags are automatically stored on stack, whereas in the INTEL 8080 only the program counter is stored automatically on stack.

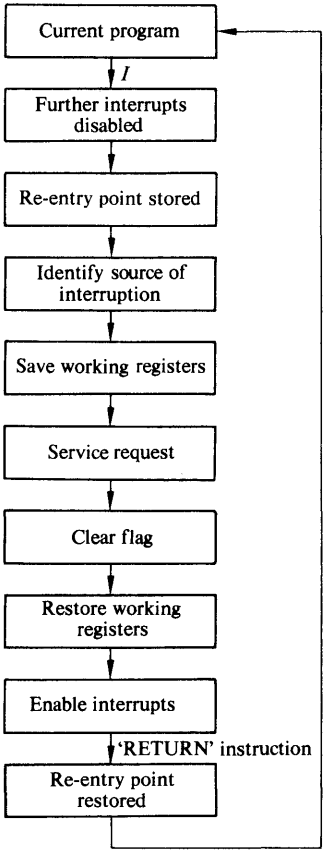


Figure 5.3.

Clearly, in cases like this it is left to the user to store all additional information which is needed to resume the interrupted program.

Because the interrupt cycles of present day microprocessors are different, the configuration of the interrupt circuit of each microprocessor will be unique. As we shall see in section 5.4 of this chapter, the design of interrupt circuits presents no special difficulty, if the interrupt cycle of the microprocessor in question is understood. However, before we discuss in detail the design and implementation of interrupt circuits, it is essential for the reader to have a clear understanding of flags and flag sorters, which we discuss next.

5.3 FLAGS AND FLAG SORTERS

Flags

A flag is defined as a signal generated and used by a device to inform some other device that it wishes to communicate with it. The block diagram of a flag circuit with *set*, *clear*, *enable* and *disable* facilities is shown in Figure 5.4. The function of each of the four input signals is as follows. A signal on terminal e enables the circuit, whereas a signal on terminal d disables the circuit. Clearly these two signals are not applied simultaneously in practice. When the circuit is enabled, a signal on terminal k sets the flag. The flag is cleared (turned off) by a signal on terminal c. If enable and disable facilities are not needed, terminals e and d may be omitted.

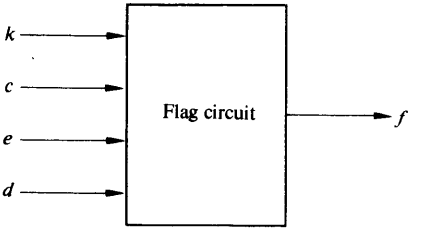


Figure 5.4.

Flag Circuits

In common with all logic circuits, the terminal characteristics of a flag circuit can be implemented by different but equivalent circuits. Below we show two possible implementations, to which we shall refer to as *flag circuit 1* and *flag circuit 2*.

Flag Circuit 1

A state diagram describing both its internal and external operations,[2] is shown in Figure 5.5. By direct reference to the state diagram, we obtain

$$\text{turn-on set of } A = B \cdot k$$

$$\text{turn-off set of } A = B \cdot \bar{k} + \bar{B} \cdot d \xrightarrow{\text{Invert}} (\bar{B} + k)(B + \bar{d})$$

$$\text{turn-on set of } B = \bar{A} \cdot e + A \cdot \bar{k}$$

$$\text{turn-off set of } B = \bar{A} \cdot d + A \cdot c + A \cdot d$$

$$= d + Ac \xrightarrow{\text{Invert}} \bar{d} \cdot (\bar{A} + \bar{c})$$

Therefore,

$$A = B \cdot k + A(\bar{B} + k)(B + \bar{d})$$

$$B = \bar{A} \cdot e + A \cdot \bar{k} + B(\bar{A} + \bar{c})\bar{d}$$

$$f = S_2 \cdot k = A \cdot B \cdot k$$

The equivalent NAND circuit is shown in Figure 5.6

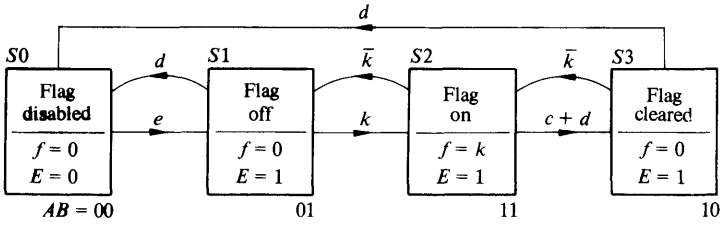


Figure 5.5.

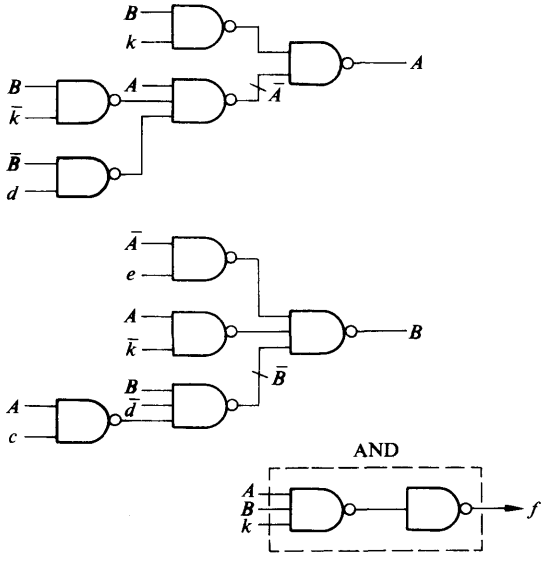


Figure 5.6.

Flag Circuit 2

If enable and disable facilities are not needed, and signal k is a pulse, the flag circuit can be implemented using a flip-flop, as shown in Figure 5.7 (see also Chapter 1, section 10).

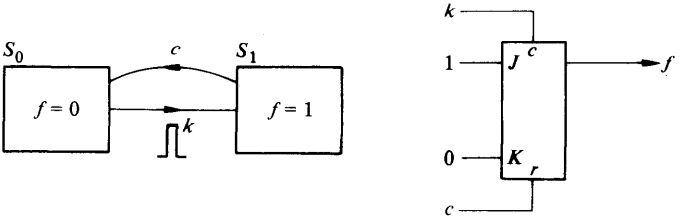


Figure 5.7.

Identification of Flags

As we mentioned earlier, the interrupt signal is generated by ORing all the flags. This signal simply informs the microprocessor that one or more devices in the system wish to communicate with it. Therefore, when the microprocessor is interrupted, the interrupt routine must identify the source of interruption. There exist two basic methods for identifying flags: *the polling method* and *the vectored method.* We shall describe each of these methods below.

The Polling Method

In this method when the microprocessor receives an interrupt signal, it sequences through the devices looking for the individual device(s) that need servicing. When it finds such a device, it stops sequencing and calls the corresponding service routine. If the interrupt signal is still on at the end of the service routine, the polling of the devices continues, otherwises the main program is resumed.

Several implementations of the polling method exist in practice. In the method we shall describe, the flags are connected to an input port from which they are read into an internal register of the microprocessor through the data bus as shown in Figure 5.8(a). The contents of the internal register are then examined one bit at a time according to the flow chart in Figure 5.8(b).

The Vectored Method

In this method the presence of flags in a system is automatically detected and the flags are identified by means of a hardware circuit, *the flag sorter*,† the

† Flag sorters are also referred to as *priority encoders*, and are commercially available as i.c. chips.

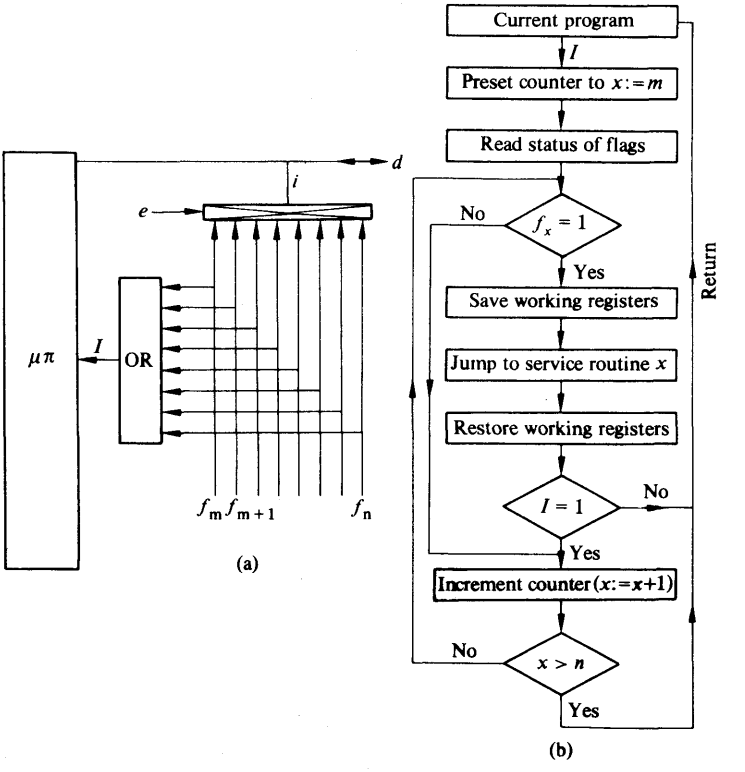


Figure 5.8.

block diagram of which is shown in Figure 5.9(a). Interrupt signal I is generated, as in all systems, by ORing the input flags. The identity of the flag is generated in a specified binary code. We shall use the true binary (8-4-2-1) code, unless we specify otherwise.

The design and implementation of flag sorters is straightforward and should present no difficulty to the reader who possesses a working knowledge of logic design, outlined in Chapter 1. We shall demonstrate the steps by designing and implementing two-flag, eight-flag and 64-flag sorters. We shall arbitrarily assume that the higher the flag number, the higher its priority.

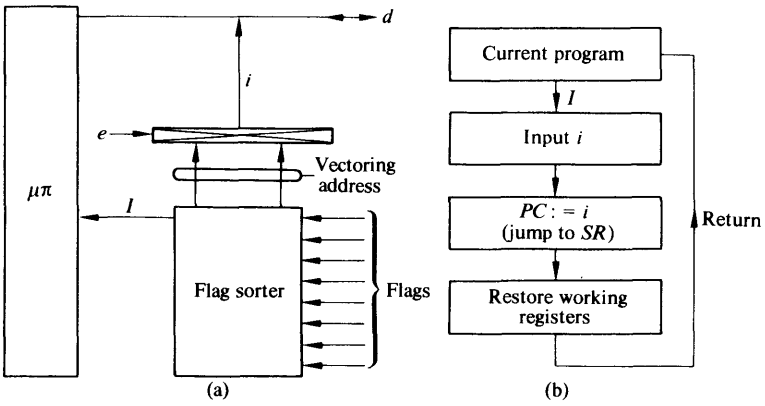


Figure 5.9.

A Two-Flag Sorter

The block diagram of a two-flag sorter is shown in Figure 5.10(a). Its I/O (input/output) relationship is shown in the form of a truth table in Figure 5.10(b). By direct reference to this table, we obtain the following equations

$$I = f_0 + f_1$$

$$A = f_1$$

Their circuit implementation is shown in Figure 5.10(c).

An Eight-Flag Sorter.

The block diagram of an eight-flag sorter is shown in Figure 5.11(a). Its I/O relationship is shown in the form of a truth table in Figure 5.11(b). By direct reference to this table, we obtain the following equations.

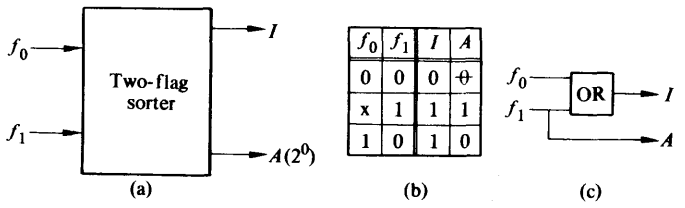


Figure 5.10.

$$I = f_0 + f_1 + f_2 + f_3 + f_4 + f_5 + f_6 + f_7$$

$$A = f_7 + \bar{f}_7 \bar{f}_6 f_5 + \bar{f}_7 \bar{f}_6 \bar{f}_5 \bar{f}_4 f_3 + \bar{f}_7 \bar{f}_6 \bar{f}_5 \bar{f}_4 \bar{f}_3 \bar{f}_2 f_1$$

$$= f_7 + \bar{f}_6 f_5 + \bar{f}_6 \bar{f}_4 f_3 + \bar{f}_6 \bar{f}_4 \bar{f}_2 f_1$$

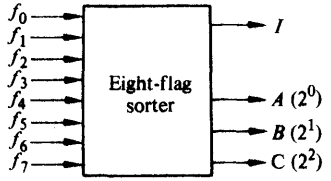
$$B = f_7 + \bar{f}_7 f_6 + \bar{f}_7 \bar{f}_6 \bar{f}_5 \bar{f}_4 f_3 + \bar{f}_7 \bar{f}_6 \bar{f}_5 \bar{f}_4 \bar{f}_3 f_2$$

$$= f_7 + f_6 + \bar{f}_5 \bar{f}_4 f_3 + \bar{f}_5 \bar{f}_4 f_2$$

$$C = f_7 + \bar{f}_7 f_6 + \bar{f}_7 \bar{f}_6 f_5 + \bar{f}_7 \bar{f}_6 \bar{f}_5 f_4$$

$$= f_7 + f_6 + f_5 + f_4$$

As eight-input priority encoders are available commercially, the above equations will not be implemented.



(a)

INPUTS								OUTPUTS			
f_0	f_1	f_2	f_3	f_4	f_5	f_6	f_7	I	C	B	A
0	0	0	0	0	0	0	0	0	0	0	0
x	x	x	x	x	x	x	1	1	1	1	1
x	x	x	x	x	x	1	0	1	1	1	0
x	x	x	x	x	1	0	0	1	1	0	1
x	x	x	x	1	0	0	0	1	1	0	0
x	x	x	1	0	0	0	0	1	0	1	1
x	x	1	0	0	0	0	0	1	0	1	0
x	1	0	0	0	0	0	0	1	0	0	1
1	0	0	0	0	0	0	0	1	0	0	0

(b)

Figure 5.11.

A 64-Flag Sorter

The block diagram of a 64-flag sorter is shown in Figure 5.12. The flags are arranged into eight groups of eight flags, each group being allocated a flag sorter. The interrupt signals from the eight flag sorters are connected to a group selector, itself an eight-flag sorter. It operates as follows.

The group selector selects a group flag that is on, generates the system interrupt signal, *I*, and a three-bit address *DEF* which identifies the selected group. Signals *D*,*E*, and *F*, in addition to being connected to the address bus, drive a binary-to-decimal decoder. Each of the eight outputs of the decoder drives in turn the three tristates which connect the address lines of the corresponding flag sorter to the address bus, as shown in Figure 5.12.

Note that our 64-flag sorter arrangement can be used directly to accommodate less than 64 flags by simply grounding the unused flag terminals.

Clearly the modular method we used to derive a 64-flag sorter using eight-flag sorters, can be used to produce a system for handling up to 4012 flags simply by using the 64-flag sorter in Figure 5.12 as the module.

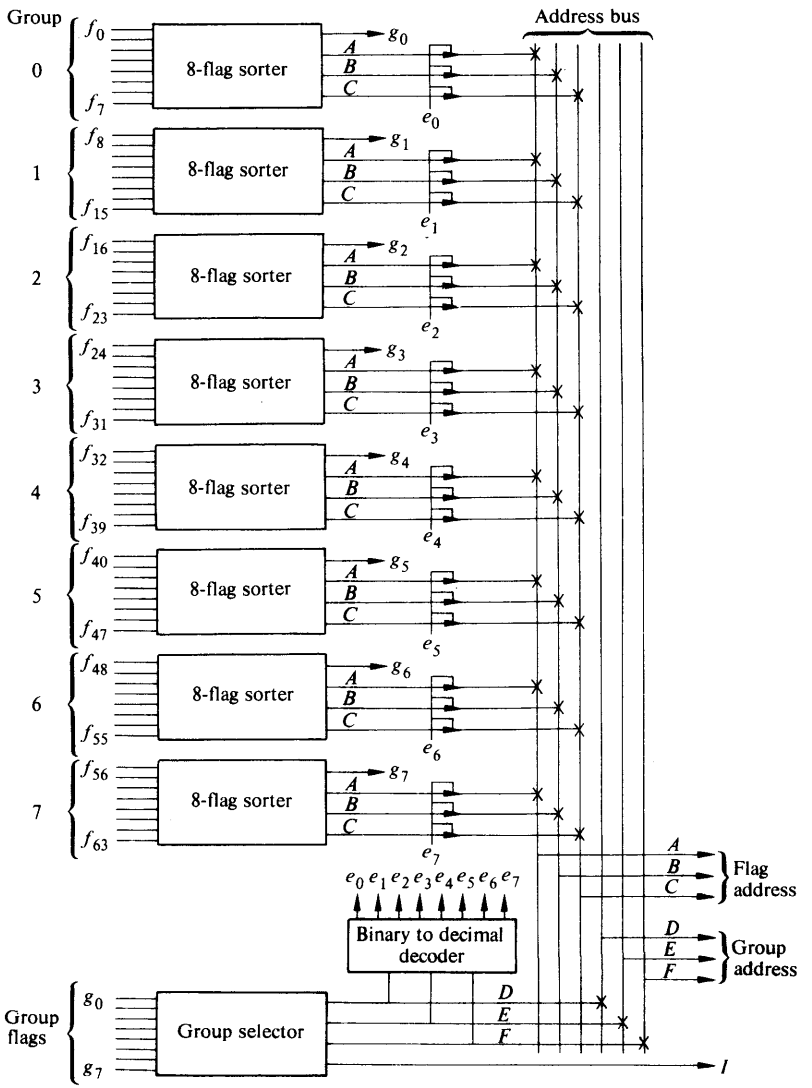


Figure 5.12.

5.4 INTEL 8080 INTERRUPT SYSTEM

In this section we shall design the interrupt system for the INTEL 8080. As we explained earlier, in order to design its interrupt circuit we must understand its interrupt cycle, which, as we shall see, is rather unique. A detailed explanation of it is given below.

The Interrupt Cycle of the INTEL 8080 [3]

Reference to its m.p.u. signal chart in Figure 2.8 shows that a logic 1 on pin 14 interrupts the program, if the interrupt terminal has not been disabled. Program interruption is indicated by a logic 1 on pin 16. There are no time constraints on the interrupt signal; it can occur at any time. The reason for this is that synchronization with the internal operation of the microprocessor is achieved by setting an internal latch with clock pulse ϕ_2 during the last state of the instruction cycle in which the interrupt request occurs.

As we already mentioned, the method used by the INTEL 8080 chip is rather unique. It is probably best understood by first examining the step-by-step execution of the *restart instruction*, the format of which is 11 *ddd* 111. The three letters *ddd* represent an octal number from 0 to 7. As in the case of an I/O instruction, the restart instruction is executed in three machine cycles.

Let us assume that the restart instruction has been loaded *somehow* into the instruction register *IR* in Figure 5.13. During the following two machine cycles

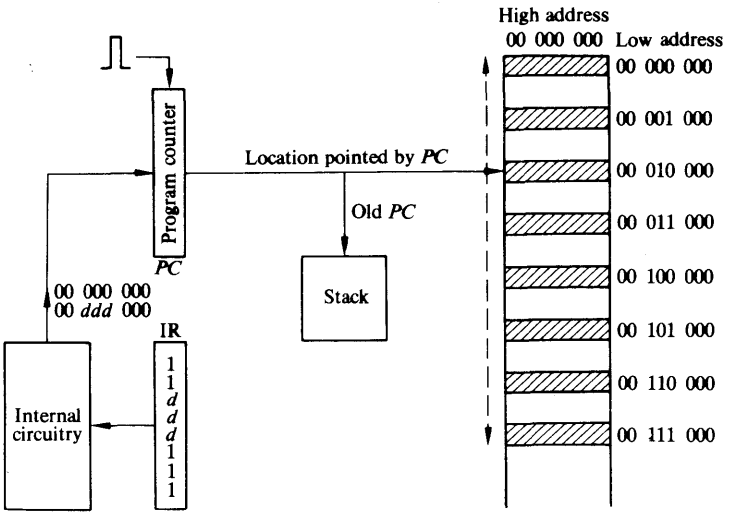


Figure 5.13.

the contents of the program counter *PC* are pushed on stack, eight bits at a time. In parallel to the stacking operation, the contents of the instruction register (11 *ddd* 111) drive some internal circuitry in the m.p.u. chip, which responds by generating sequentially two eight-bit words shown below

00 000 000, and

00 *ddd* 000.

As each set of eight bits in the program counter is pushed on stack, the two eight-bit words generated internally 00 000 000 and 00 *ddd* 000 are moved into *PC*, 00 000 0000 in the portion that holds the high address and 00 *ddd* 000 in the portion that holds the low address. The implication of this is that the next instruction to be executed will be fetched from one of the eight locations listed below.

00 000 000	(000_8)	00 000 000	(000_8)	, when $ddd = 000$
00 000 000	(000_8)	00 001 000	(010_8)	, when $ddd = 001$
00 000 000	(000_8)	00 010 000	(020_8)	, when $ddd = 010$
00 000 000	(000_8)	00 011 000	(030_8)	, when $ddd = 011$
00 000 000	(000_8)	00 100 000	(040_8)	, when $ddd = 100$
00 000 000	(000_8)	00 101 000	(050_8)	, when $ddd = 101$
00 000 000	(000_8)	00 110 000	(060_8)	, when $ddd = 110$
00 000 000	(000_8)	00 111 000	(070_8)	, when $ddd = 111$

As we saw in the previous section, the three variables *ddd* in the restart instruction, are generated by a flag sorter.

Later we shall see how the restart instruction can be used to interrupt the INTEL 8080 for emergency situations when interrupts are disabled. This feature, which is analogous to the non-maskable interrupts in the Motorola 6800 chip, is essential in process control applications, in medical and other similar high-risk environments.

Let us now return to the interrupt cycle of the INTEL 8080. This resembles very closely a normal fetch instruction shown in Figure 2.2 and 2.4, with the exception that (1) An *INTA* (interrupt acknowledge) status signal is generated, (2) The program counter is not incremented, and (3) The interrupt terminal is disabled.

If during the *INTA* ·*DBIN* interval we disconnect the memory from the data bus and 'jam' onto it a restart instruction, as shown in Figure 5.14, the program will vector to one of the eight locations shown in Figure 5.13 depending on the output of the flag sorter, *ddd*. For example, if the output of the flag sorter is 010

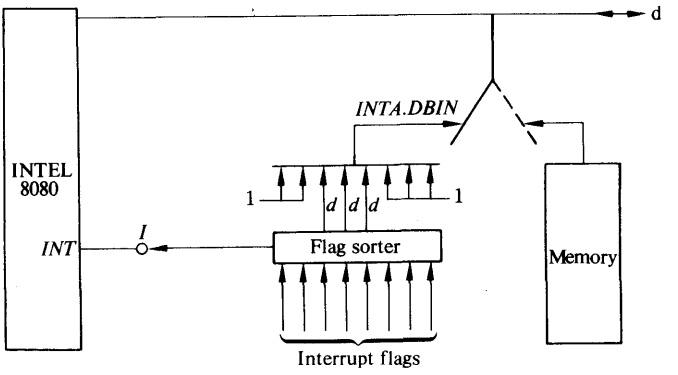


Figure 5.14.

the program will branch to location 00 010 000 (020_8) in memory with high address 00 000 000 (000_8).

In existing systems, typically but not necessarily, high address 00 000 000 (000_8) specifies a ROM. Since service routines written by users must reside in RAMS (you cannot write in ROMS), the locations in ROM specified by 00 *ddd* 000 contain JUMP instructions to specific locations in RAM, as shown in Figure 5.15. Unless we specify otherwise, we shall assume that the RAM address in our case is 00 000 011 (003_8) and that the locations in RAM to which the program jumps to are 000_8 to 070_8. From the user point of view, this means that the result of an interrupt signal is to cause the program to vector to one of

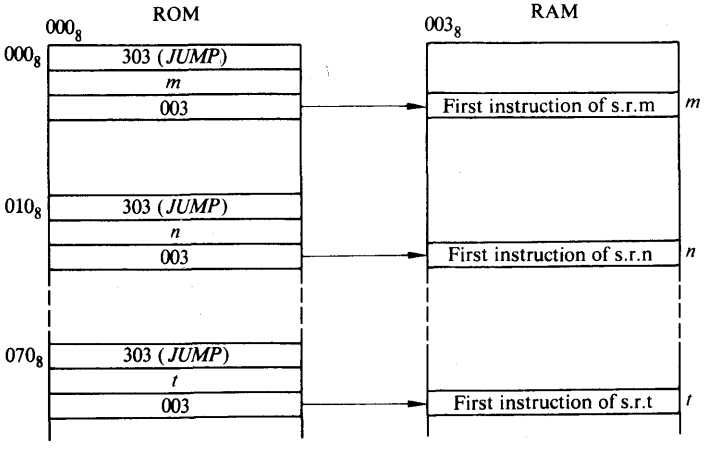


Figure 5.15.

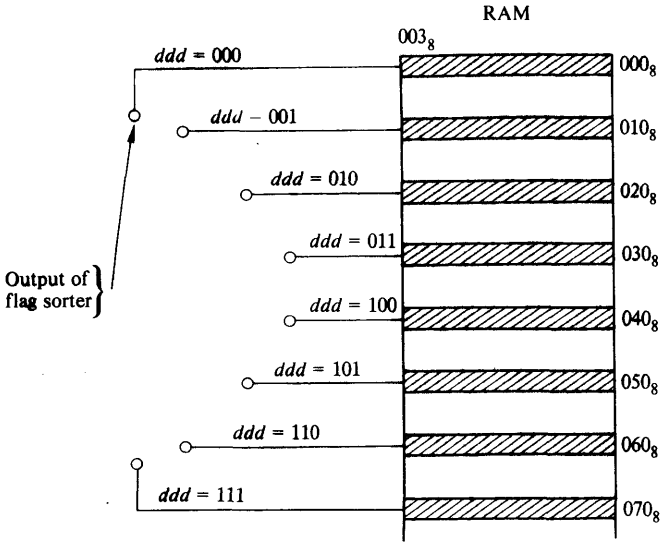


Figure 5.16.

eight locations in RAM, depending on the values of the *ddd* signals, as shown in Figure 5.16. Any of the eight locations can be used to store the first instruction of a routine designed to service an interrupting device. If a service routine contains more than eight instructions, the programmer can use a 'jump' instruction to move to a different location in memory.

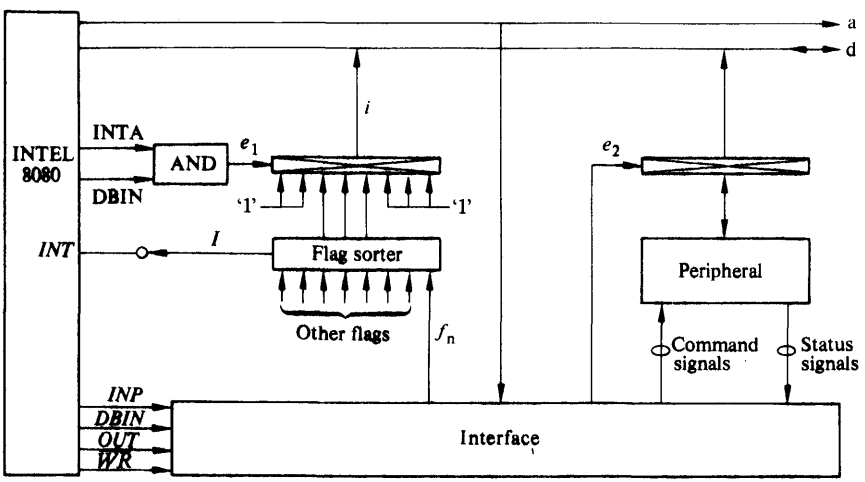


Figure 5.17.

The program counter is restored to its pre-interrupt value by executing a *return instruction.* Working registers are restored by executing the appropriate 'pop' instructions.

The block diagram of a microprocessor interrupt system using the INTEL 8080 can now be drawn as in Figure 5.17.

5.5 EMERGENCY INTERRUPTS FOR THE INTEL 8080

One of the essential features in any system when operating in medical, industrial and other high-risk environments, is its ability to respond with the least possible delay to emergency situations. In the case of the INTEL 8080, as we have seen earlier on, when it enters an interrupt cycle, its interrupt terminal is automatically disabled. To be re-enabled an enable-interrupt (*EI*) instruction must be executed. Until such an instruction is implemented the microprocessor will not respond to external interrupts—that is emergency signals are ignored during this period. This time interval can be dangerously extended if, either due to some oversight on the part of the programmer or due to carelessness or ignorance, he fails to enable the interrupts at the beginning of a service routine to allow for such emergencies.

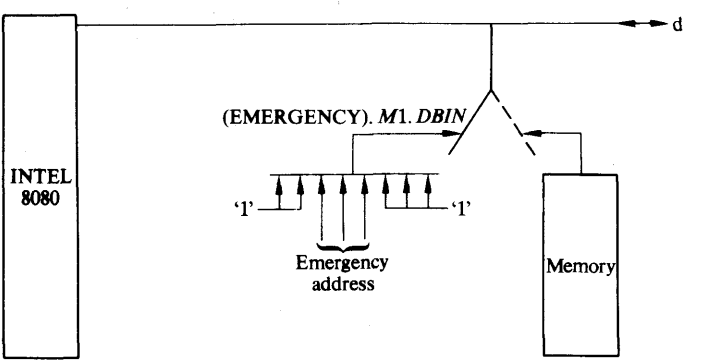


Figure 5.18.

Such risks can be eliminated in practical situations by 'jamming' onto the data bus a restart instruction when an emergency arises with a specified value for *ddd*, as shown in Figure 5.18. This will cause the program to vector almost instantly to an emergency service routine, irrespective of whether the interrupt terminal is disabled or not.

5.6 MOTOROLA 6800 INTERRUPT SYSTEMS

As we shall see later, in the case of the M6800 we can design two interrupt systems. Before we do so, as with all microprocessors, we must have a clear understanding of its interrupt cycle. This is described next.

The Interrupt Cycle of the MOTOROLA 6800 [4]

Reference to its m.p.u. chart in Figure 2.9 shows that the M6800 can be interrupted by a logic '0' on pin 14, if interrupts are not disabled. As in the case of the INTEL 8080, there are no time constraints on the interrupt signal; it can occur at any time. The reason for this is that synchronization with the instruction in progress is achieved by setting an internal latch with clock pulse ϕ_2 during the last state of the instruction cycle in which the interrupt cycle occurs.

Its interrupt sequence is summarized below.

1. The current instruction is completed,
2. Further interrupts are disabled,
3. The m.p.u. status is pushed automatically on stack in the following order

PC_L
PC_H
IR_L
IR_H
ACCA
ACCB
CC

4. The program counter is next loaded with the contents of memory locations *FFF*8 (PC_H) and *FFF*9 (PC_L). These contents, as we have already explained, specify the address of the first instruction of the interrupt routine.

The reader's attention is drawn to the fact that at this point the source of interruption has not been identified.

To resume the interrupted program at the end of the interrupt routine the user executes a *return from interrupt* (*RTI*) instruction, whose hexadecimal code is 3*B*. Execution of this instruction restores the pre-interrupt m.p.u. status by 'popping' from stack the m.p.u. registers in the reverse order.

Because in the case of the M6800 the source of interruption can be identified using either the polling method or the vectored method, described in section 5.3.3, we have the choice of designing two interrupt systems, each using a different method of flag identification, as shown next.

Interrupt System 1

In this system we use the polling method to identify the source of interruption. Its block diagram is shown in Figure 5.19. Its step-by-step operation is flowcharted in Figure 5.8(b).

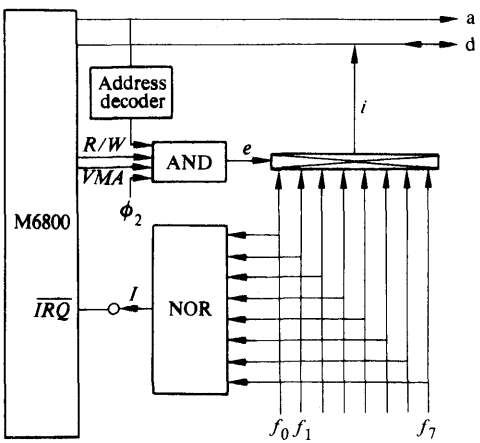


Figure 5.19.

Interrupt System 2

In system 2 we use the vectored method to identify the source of interruption. Its block diagram is shown in Figure 5.20. Its step-by-step operation is flowcharted in Figure 5.21.

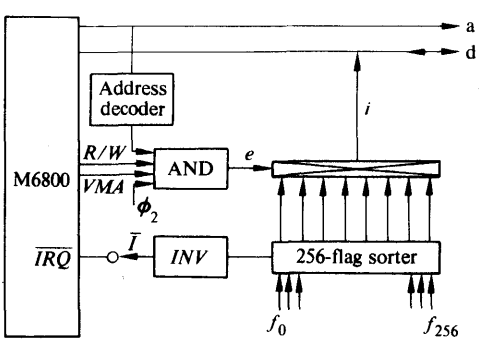


Figure 5.20.

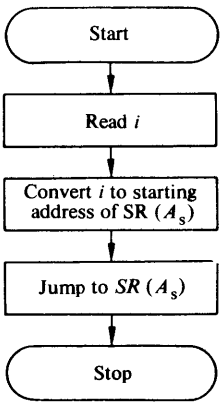


Figure 5.21.

5.7 EMERGENCY INTERRUPTS FOR THE MOTOROLA 6800

Emergency interrupts are handled by the non-maskable interrupt input on pin 39 (see Figure 2.9).

5.8 PROBLEMS AND SOLUTIONS

In this section we demonstrate our design steps by means of problems and fully-worked out solutions. The reader's attention is drawn to the fact that, although we use the INTEL 8080 and the Motorola 6800 to implement our designs, our procedures apply to all types of microprocessors. Specifically, it should be noted that the first three steps in the design are executed without reference to the microprocessor.

Problem 1 *An event-counter*

Pulses representing events arrive randomly on line q in Figure 5.22. Design an interrupt system that allows a print-out of the event-count to be produced each time a manual switch m is activated. Activation of the switch, which can be assumed to be infrequent, resets the count.

Implement you design using

(i) the INTEL 8080, and
(ii) the MOTOROLA 6800.

I/O addresses available to the designer are 003_8, and 004_8 for the INTEL 8080, and 2000_{16}, 8004_{16} and 8006_{16} for the M6800.

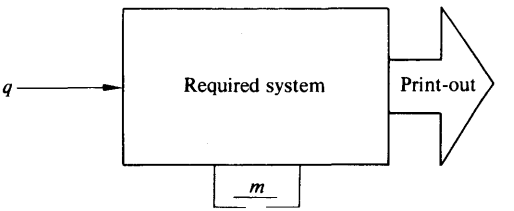


Figure 5.22.

8080 SOLUTION

Step 1 *Aim of the design*

The aim of the design is to obtain an event-count on request, using the interrupt mode, as shown in Figure 5.23.

Step 2 *Device characteristics*

The block diagram of the interrupt system using the INTEL 8080 is shown in Figure 5.17 and the microprocessor I/O signals in Figure 4.7.

The terminal characteristics of the printer are shown in Figure 5.24.

Step 3 *System design*

The block diagram of our general solution is shown in Figure 5.25. Its operation, flowcharted in Figure 5.26, is as follows. Each time a pulse arrives on line q, we shall interrupt the current program and execute a service routine

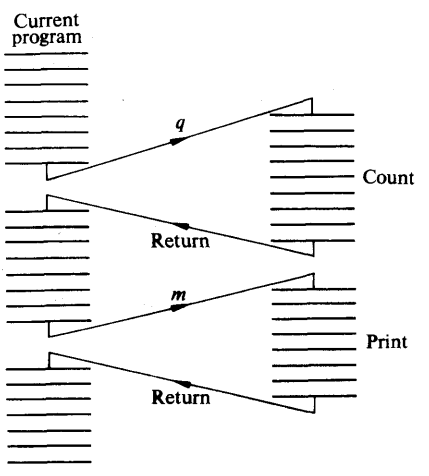
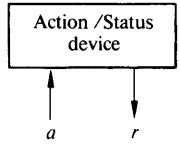


Figure 5.23.

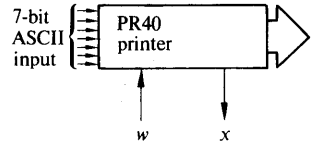
Action/status devices



Terminal a. *A* 0 to 1 transition on this line activates device.

Terminal r. Status signal *r* is 1 when the device is ready, otherwise $r = 0$. Activation of the device is not possible when $r = 0$.

Printer



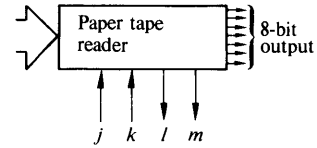
Terminal w. A negative-going pulse of one microsecond minimum duration loads the character into the buffer, or activates the printing mechanism if either the buffer is full or the ASCII code† for carriage return '015' in octal is being input.

Non-ASCII characters are ignored.

Terminal x. Status signal x equals 1 when the printer is ready and the *w* line is at 1.

†See Figure 3.23 (p.74).

Reader



Terminal j. A ground on this terminal causes the tape to move left.

Terminal k. A ground on this terminal causes the tapes to move right.

Terminal l. Status signal *l* is 1 when sprocket hole is under the read head, otherwise $l = 0$.

Terminal m. A logic 1 indicates that the reader is available for the next drive pulse.

To stop tape on next character remove drive within 1 msec after the leading edge of signal *l*. Its minimum duration is 1 μsec.

Figure 5.24.

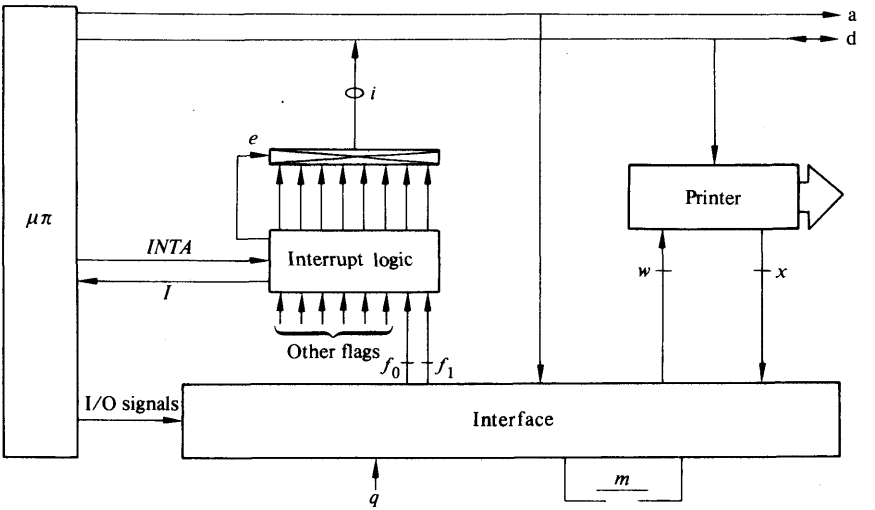


Figure 5.25.

that increments a counter by 1. Similarly, when switch m is activated, we shall interrupt the current program and execute another service routine which produces a print-out of the event-count and resets the counter to 0.

Step 4 *Hardware design*

Reference to Figure 5.27, the block diagram of our INTEL 8080 solution, indicates that the signals to be generated by our interface are

1. The two flags, f_0 and f_1, and
2. The print pulse, w.

If no enable and disable facilities are required for our flag signals, they can be implemented using flag circuit 2, described in section 5.2 of this chapter. If we use the q pulses to set flip-flop f_0 in Figure 5.28, and the activation/release of switch m to set flip-flop f_1, then the equations of our interface signals are

$$c_0 = q,$$

$$r_0 = \overline{OUT \cdot WR \cdot A_{003}},$$

$$c_1 = m,$$

$$r_1 = w \text{ and}$$

$$w = \overline{OUT \cdot WRA_{004}}$$

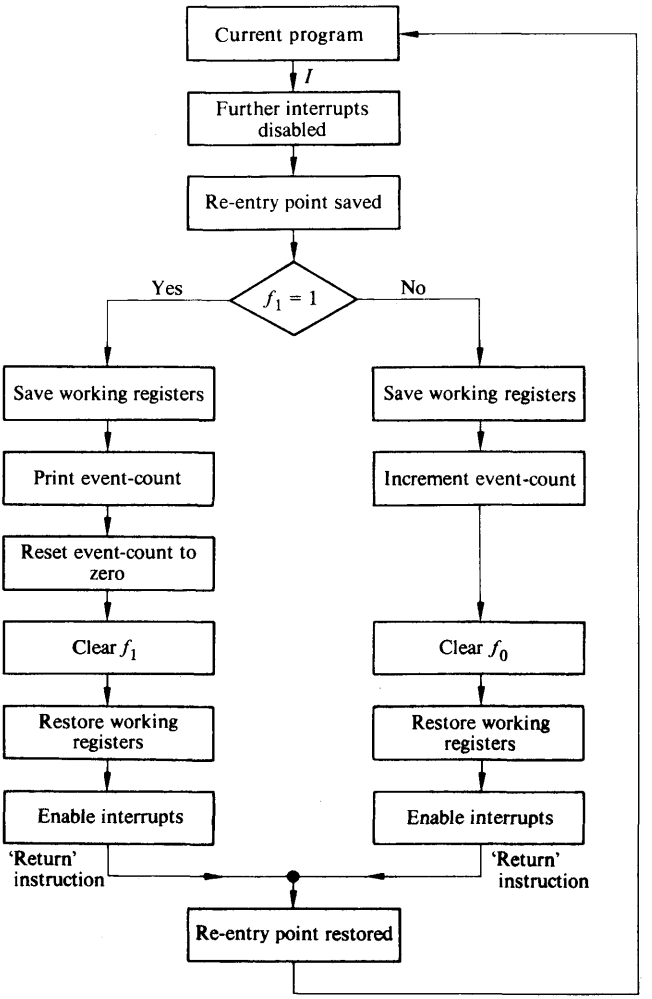


Figure 5.26.

If we further make the assumption that f_0 and f_1 are the only two flags in the system, a two-flag sorter will suffice. As we saw in section 5.3 of this chapter, such a circuit consists of an OR gate (Figure 5.10). Under these conditions the hardware component of our system is shown in Figure 5.28.

Step 5 *Software design*

The programming flow charts for the INTEL 8080 are shown in Figure

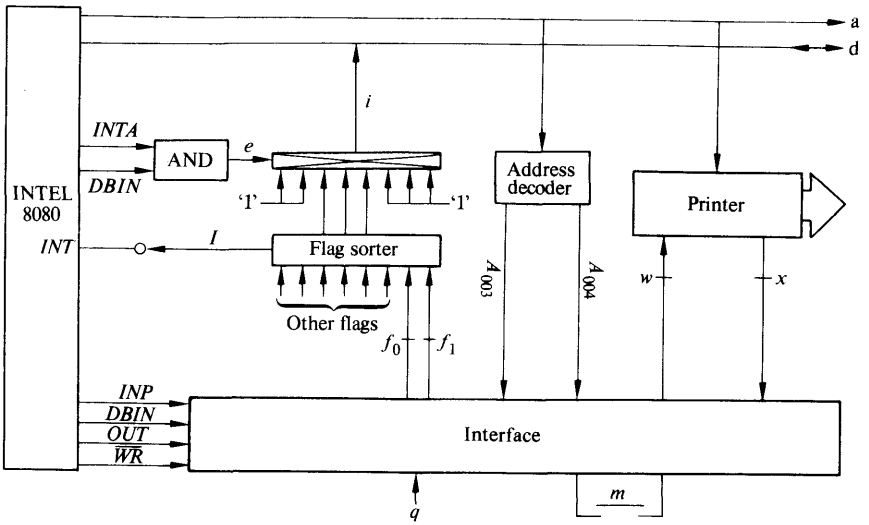


Figure 5.27.

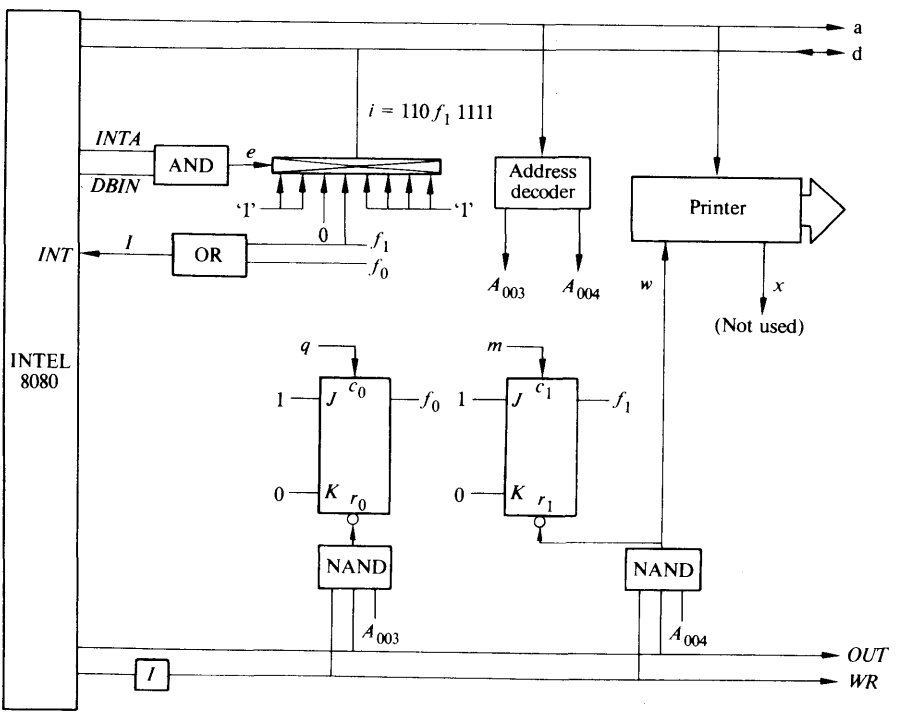


Figure 5.28.

5.29.† By direct reference to them and to the instruction set in Figure 3.28 (or to the chart in Figure 3.36), we obtain the octal and hexadecimal listings shown below.

Octal address		*Octal listing*	*Hex listing*	*Mnemonics*		*Comments*
			Service Routine 0 (Count)			
H	*L*					
003	010	365	*F5*	*PUSH PSW*		Save *AC* and flags.
	011	004	04	*INR B*		Increment *B*.
	012	323	*D3*	*OUT*	⎫	I/O cycle to clear
	013	003	03		⎭	flag f_0.
	014	361	*F1*	*POP PSW*		Restore *AC* and flags.
	015	373	*FB*	*EI*		Enable interrupts.
	016	311	*C9*	*RET*		Return.
			Service Routine 1 (Print)			
	030	365	*F5*	*PUSH PSW*		Save *AC* and flags.
	031	170	78	*MOV A,B*		Move *B* into *A*.
	032	323	*D3*	*OUT*	⎫	I/O to load printer and
	033	004	04		⎭	clear f_1.
	034	076	*3E*	*MVI A*	⎫	Load *AC* with *ASCII* code
	035	015	*0D*	015	⎭	for carriage return.
	036	323	*D3*	*OUT*	⎫	Activate printer.
	037	004	04		⎭	
	040	006	06	*MVI B*	⎫	Clear register *B* (our
	041	060	30		⎭	counter).
	042	361	*F1*	*POP PSW*		Restore *AC* and flags.
	043	373	*FB*	*EI*		Enable interrupts.
	044	311	*C9*	*RET*		Return.

6800 SOLUTION

Step 1
Step 2
Step 3
} Same as for 8080 solution

† Reference was made to Figure 5.3.

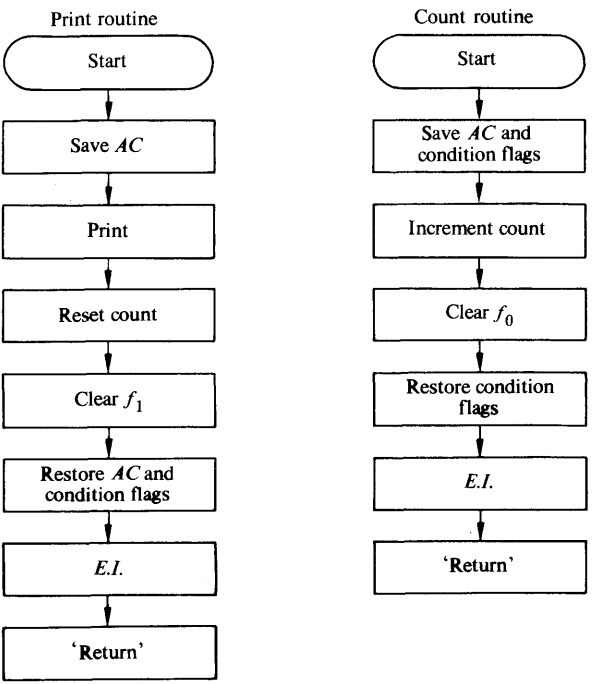


Figure 5.29.

Step 4 *Hardware design*

In the case of the M6800 the designer has the option to use a polling or a vectored interrupt system. In this solution we shall use the polling system described in section 5.3 of this chapter. The block diagram of our solution in this case is shown in Figure 5.30. Reference to it indicates that the signals to be generated by our interface are

1. Two flag signals, f_0 and f_1, to initiate the *count* and *print* routines,
2. The enable signal e for the I/O port, and
3. The w signal to activate the printer.

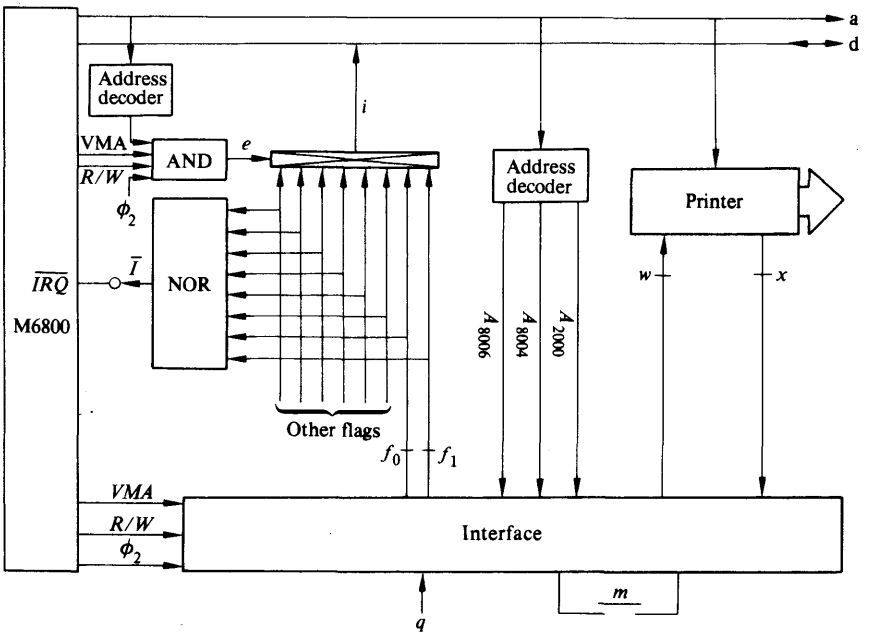


Figure 5.30.

As before, we shall assume that f_0 and f_1 are the only two flags in the system. This allows us to generate the interrupt signal and the address of the interrupting device using a single AND gate, as explained in section 5.2 of this chapter. If we further assume that no enable/disable facilities are required for our two flags, they can be generated by the two *JKFF*s, as shown in Figure 5.31. A pulse on line q sets the *count* flip-flop, turning f_0 on; similarly a pulse on line m sets the *print* flip-flop, turning f_1 on. If hexadecimal addresses 2000 and 8006 are used to clear the two flags, the equations of the *clear* signals are

$$\overline{r_0} = VMA \cdot \overline{R/W} \cdot A_{2000} \cdot \phi_2, \text{ and}$$

$$\overline{r_1} = VMA \cdot \overline{R/W} \cdot A_{8006} \cdot \phi_2$$

NAND gates 1 and 2 in Figure 5.31 implement these two signals.

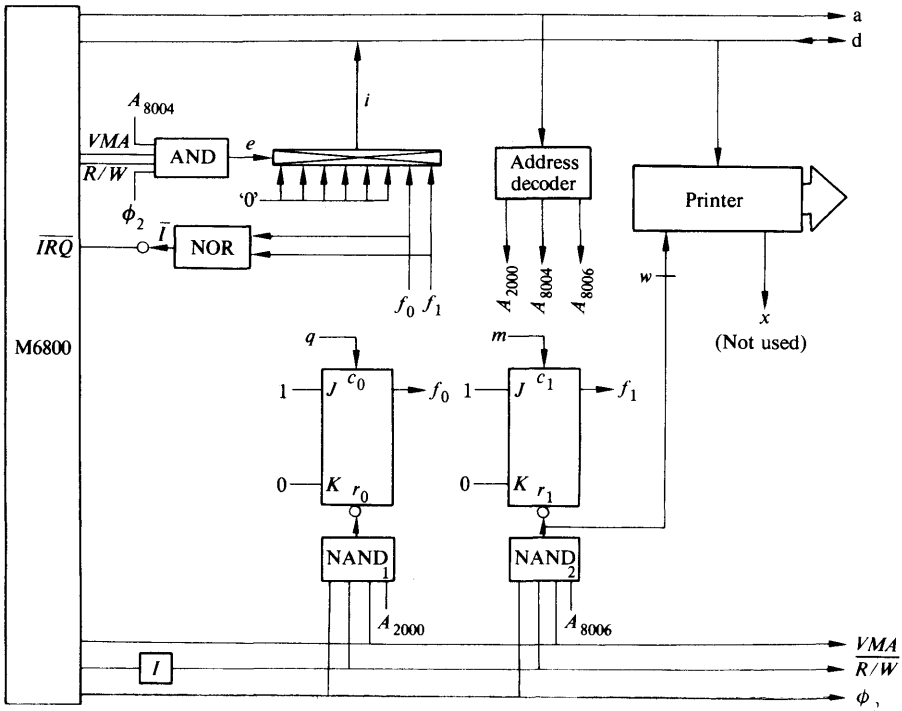


Figure 5.31.

In this system we can activate the printer when we clear flag f_1. Because the printer is activated and the flag is reset (turned off) on a 1 to 0 signal transition, these two operations may occur at the same time, making

$$w = r \qquad \text{(See Figure 5.31).}$$

Note the degree of similarity between the INTEL 8080 and the M6800 systems.

Step 5 *Software design*

The programming flow chart of our solution is shown in Figure 5.32.† By direct reference to it and to the M6800 programming chart in Figure 3.30 (or to its instruction set in Figure 3.31), we obtain the hexadecimal listing of our service routine.

† Reference was made to Figure 5.3.

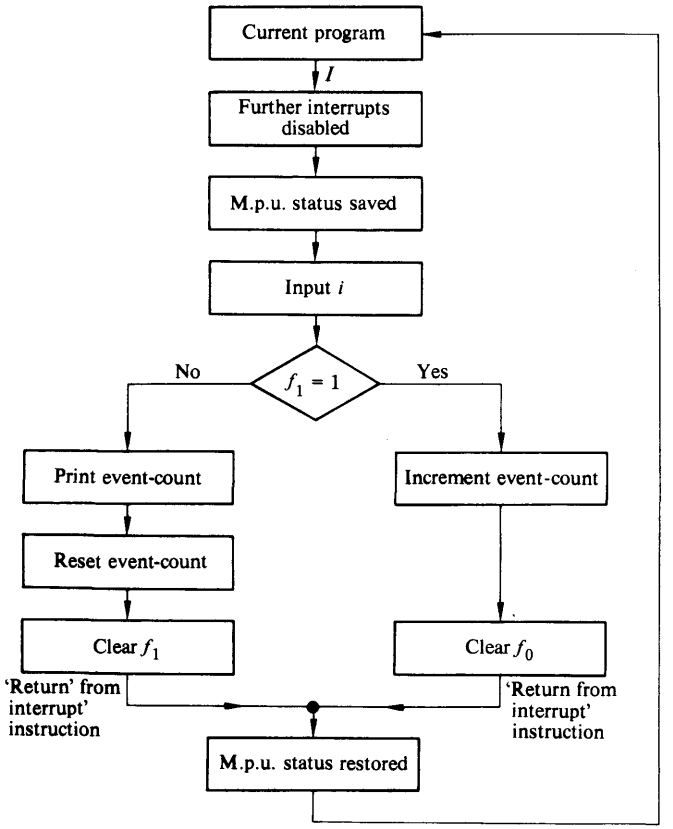


Figure 5.32.

	Hex address	*Hex listing*	*Mnemonics*	*Comments*
00	50	*B6*	*LDA A*	Input address from flag sorter.
	51	80		
	52	04		
	53	87	*RAR A*	Rotate right *A*
	54	25	*BCS L1*	Jump to count routine, if carry flag is set.
	55	07		
	56	*F6*	*LDA B*	Move counter into *B*
	57	00		
	58	*FF*		
	59	*F7*	*STA B*	Print and clear flag f_1.
	5*A*	80		
	5*B*	06		
	5*C*	3*B*	*RTI*	Return.
L1:	5*D*	7*C*	*INC*	Increment counter
	5*E*	00		
	5*F*	*FF*		
	60	*F7*	*STA B*	Clear flag f_2.
	61	20		
	62	00		
	63	3*B*	*RTI*	Return.

Problem 2 *RAM to printer interface*

Design and implement an interface between an eight-bit microprocessor and a digital printer.

Use the interrupt mode to implement your design, which is to be verified using (i) the INTEL 8080, and (ii) the MOTOROLA 6800. I/O addresses available are 004_8, 005_8 and 006_8 for the INTEL 8080, and 2000_{16}, 4000_{16} and 8004_{16} for the M6800.

8080 SOLUTION

Step 1 *Aim of the design*

The aim of the design is to enable the programmer to transfer a block of data, which is stored in consecutive locations in RAM, into an acceptor, using the interrupt mode, as shown in Figure 5.33.

Step 2 *Device characteristics*

The block diagram of the interrupt system using the INTEL 8080 is shown in Figure 5.17 and the microprocessor I/O signals in Figure 4.7.

The terminal characteristics of the printer are shown in Figure 5.24.

Step 3 *System design*

The block diagram of our solution is shown in Figure 5.34. Its operation, flow-charted in Figure 5.35, is as follows. At each program interruption the next character to be printed is transferred from the next location in RAM into the accumulator. From the accumulator it is output onto the data bus and the printer is activated. After the character has been printed and the printer becomes ready, the next interruption is initiated by generating flag f_7.

When the last character of the block has been printed, the programmer disables the interrupt flag and no more characters are printed. The next block transfer is initiated by the programmer re-enabling the flag.

Step 4 *Hardware design*

The block diagram of our solution using the INTEL 8080 is shown in Figure 5.36. The two signals to be generated by the interface hardware are

1. The print command, w, and
2. The interrupt flag f_7.

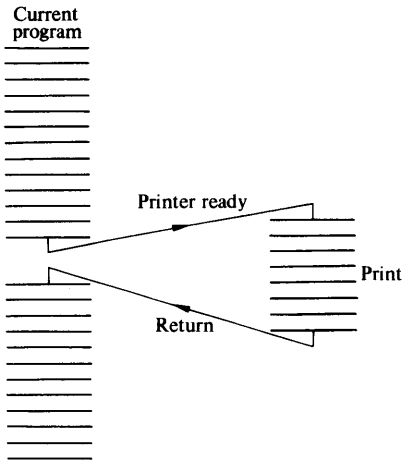


Figure 5.33.

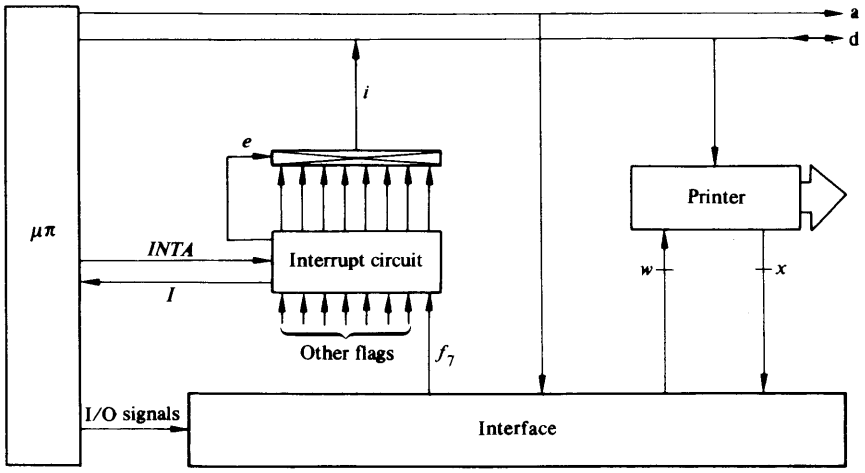


Figure 5.34.

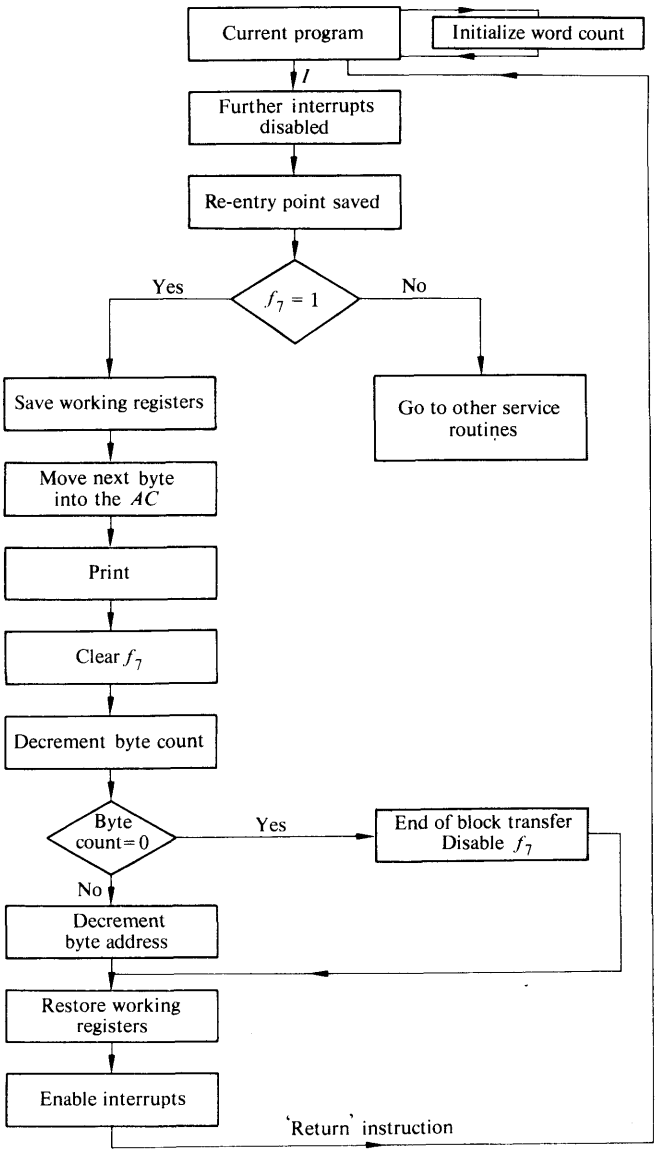

Current program
Initialize word count
I
Further interrupts disabled
Re-entry point saved
Yes
$f_7 = 1$
No
Save working registers
Go to other service routines
Move next byte into the AC
Print
Clear f_7
Decrement byte count
Byte count=0
Yes
End of block transfer
Disable f_7
No
Decrement byte address
Restore working registers
Enable interrupts
'Return' instruction

Figure 5.35.

The most straightforward method of generating signal w is to decode an I/O instruction with one of the available I/O addresses, say 004_8. In such a case

$$w = \overline{OUT \cdot WR \cdot A_{004}}.$$

If the duration of signal w is less than 1 microsecond, we would have to *stretch* it using standard methods, or add to the printer a front-end logic, using the methods outlined in Appendix 1.

Flag f_7 should be turned on when the printer becomes *ready*, that is when its status signal changes to 1. It can be cleared when we activate the printer, that is when terminal w is grounded. If we use a clocked flip-flop to generate f_7, as we explained in section 5.2 of this chapter, then the equations of its clock and reset signals become

$$c_1 = \bar{x}, \text{ and}$$

$$r_1 = w$$

The flag flip-flop is shown in Figure 5.37. The reader is reminded that our flip-flops, unless we specify otherwise, switch on the trailing edge of a clock pulse and reset with a ground, that is with a logic 0.

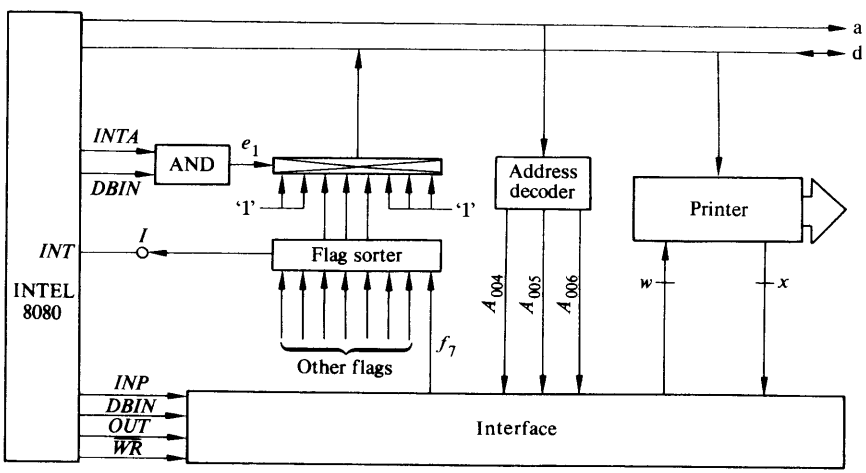


Figure 5.36.

In addition, the interface must provide the programmer with the facility to use software for enabling and disabling the interrupt flag. Such a facility can be supplied by a flip-flop, which can be turned on and off under program control. If we allocate octal addresses 005 and 006 for this purpose, the relevant equations are

$$c_0 = OUT \cdot WR \cdot A_{005}, \text{ and}$$

$$r_0 = OUT \cdot WR \cdot A_{006}$$

Their circuit implementation is shown in Figure 5.37.

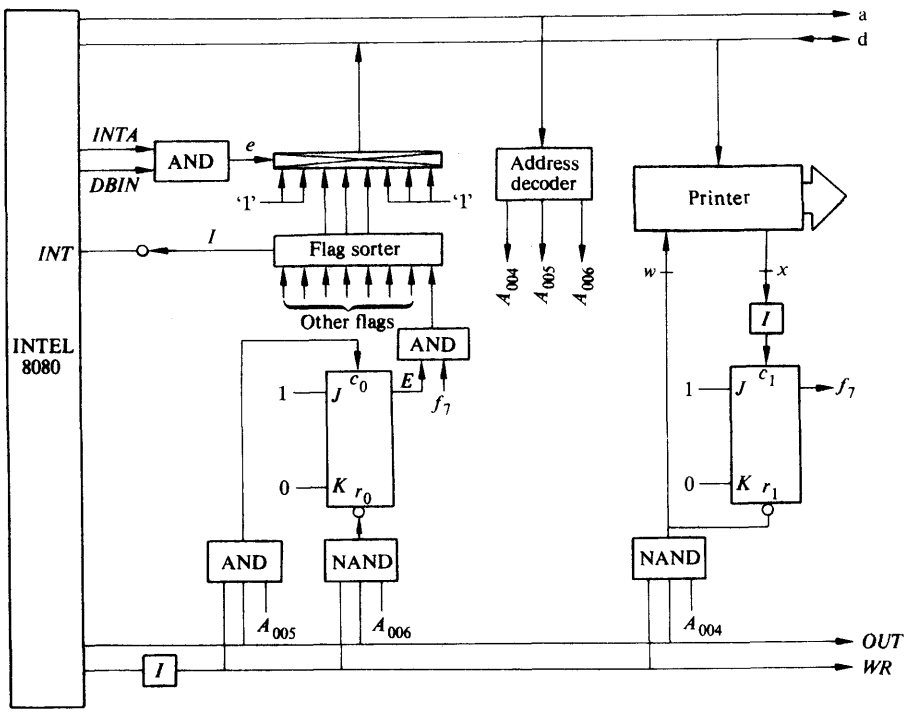


Figure 5.37.

Step 5 *Software design*

The programming flow chart of our solution is shown in Figure 5.38.† By direct reference to it and to the programming chart in Figure 3.27 (or to the instruction set in Figure 3.28), we obtain the octal and hexadecimal listings of our service routine shown below.

Octal address		*Octal listing*	*Hex listing*	*Mnemonics*	*Comments*
003	060	365	*F5*	*PUSH PSW*	Save working registers.
	1	345	*E5*	*PUSH H*	
	2	032	1*A*	*LHLD*	Move into HL the address of the next byte (contents of locations 003–056/7).
	3	056	2*E*		
	4	003	03		
	5	276	7*E*	*MOV A,M*	Move next byte into *AC*.
	6	323	*D3*	*OUT*	Output character and clear flag.
	7	004	04		
	070	053	2*D*	*DCX H*	Decrement HL and copy into locations 003–056/7. This is the address in memory where the next byte resides.
	1	042	22	*SHLD*	
	2	056	2*E*		
	3	003	03		
	4	041	21	*LXIH*	Load HL with the address of the location in memory where the byte count is stored.
	5	055	2*D*		
	6	003	03		
	7	065	35	*DCRM*	Decrement byte count.
	100	302	*C2*	*JNZ*	If the byte count is not zero, jump to location *L1*.
	1	105	45	*L1*	
	2	003	03		
	3	323	*D3*	*OUT*	Disable f_7 if byte count is zero.
	4	006	06		
L1	5	341	*E1*	*POP H*	Restore working registers.
	6	361	*F1*	*POP PSW*	
	7	373	*F8*	*EI*	Enable further interrupts and return.
	10	311	*C9*	*RET*	

6800 SOLUTION

Step 1
Step 2
Step 3

Same as for the 8080 solution.

† Reference was made to Figure 5.3.

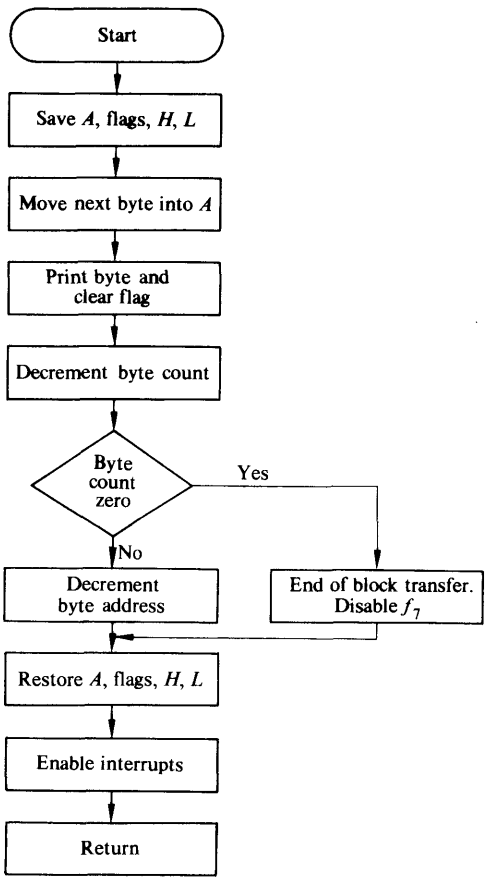


Figure 5.38.

Step 4 *Hardware design*

Reference to our system block diagram in Figure 5.39 shows that the signals to be generated by our interface are

1. Print command, w, and
2. Interrupt flag f_7.

If we allocate hexadecimal address 8004 to the printer, then

$$w = A_{8004}' VMA \cdot \overline{R/W} \cdot \phi_2, \text{ see NAND gate in Figure 5.40}$$

If the duration of signal w is less than 1 microsecond, we would have to

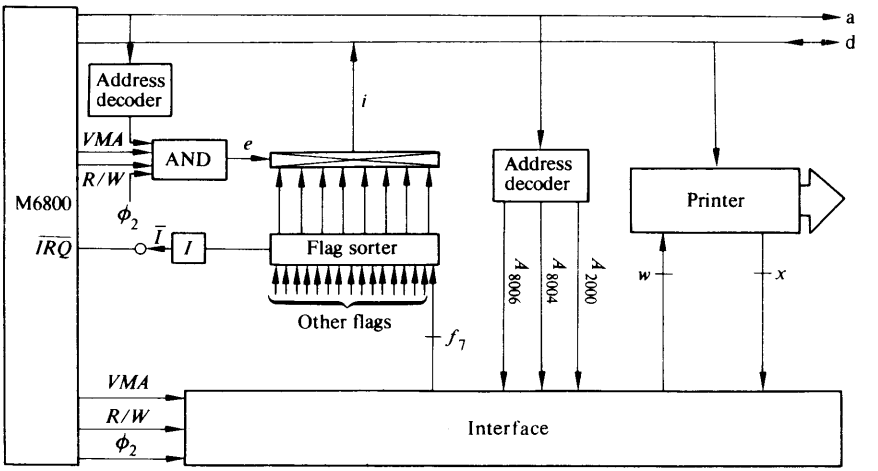


Figure 5.39.

'stretch' it using standard methods, or add a front-end logic to the printer, using the methods outlined in Appendix 1.

Flag f_7 should be turned on when the printer becomes 'ready', that is when its status signal x changes to 1. It can be cleared when we activate the printer, that is when terminal w is grounded. If we use a clocked flip-flop to generate f_7, as we explained in section 5.2 of this chapter, then the equations of its clock and reset signals become

$$c_1 = \bar{x}, \text{ and}$$

$$r_1 = w.$$

The flip-flop is shown in Figure 5.38. The reader is reminded that our flip-flops, unless we specify otherwise, switch on the trailing edge of a clock pulse and reset with a ground, that is a logic 0.

In addition, the interface must provide the programmer with the facility to use software for enabling and disabling the flag. The most straightforward method is to introduce an enable flip-flop, which can be turned on and off under program control. If we allocate hexadecimal addresses 2000 and 4000 for this purpose, then the equations of flip-flop's clock and reset signals are

$$c_0 = A_{2000} VMA \cdot \overline{R/W} \cdot \phi_2, \text{ and}$$

$$\overline{r_0} = A_{4000} VMA \cdot \overline{R/W} \cdot \phi_2$$

The output of the flag flip-flop is ANDed with E to generate signal f_7 as shown in Figure 5.40.

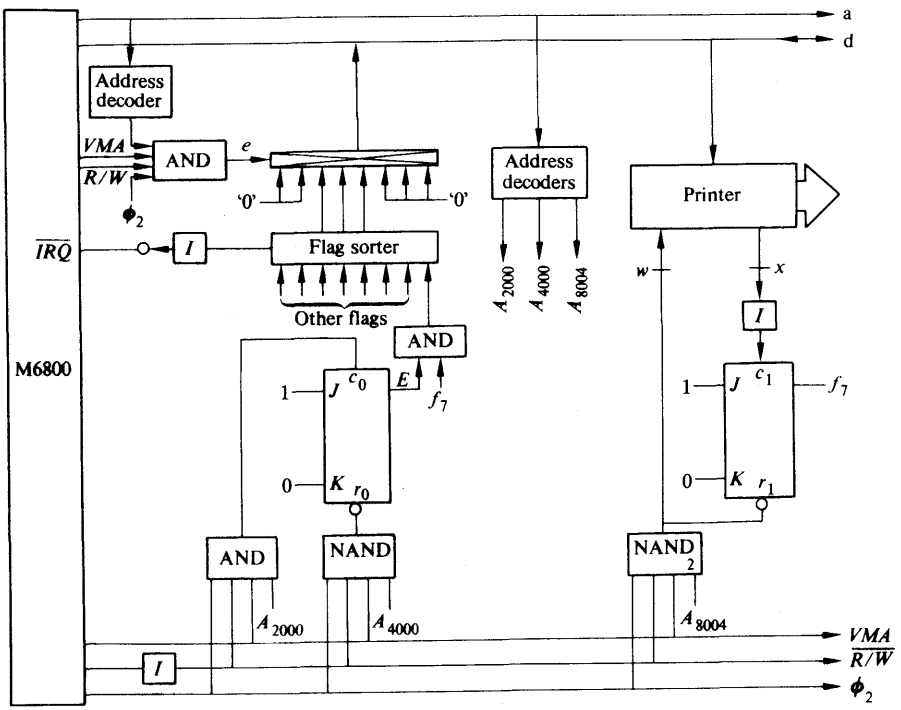


Figure 5.40.

Step 5 *Software design*

The programming flow chart of our solution is shown in Figure 5.41. By direct reference to it and to the programming chart in Figure 3.30 (or to the instruction set in Figure 3.31), we obtain the hexadecimal listing of our service routine.

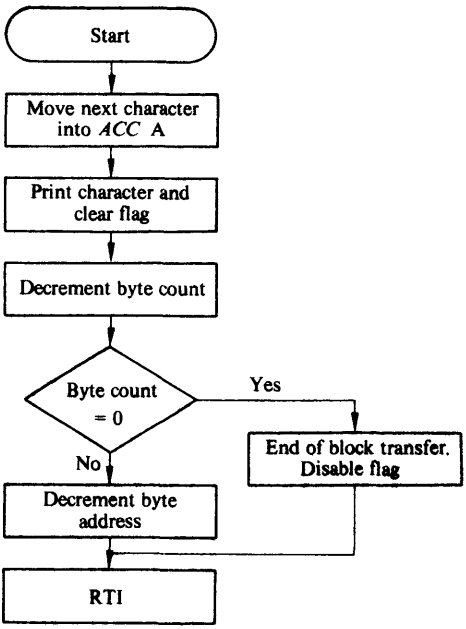


Figure 5.41.

Hex address		*Hex listing*	*Mnemonics*	*Comments*
00	5F	*n*		Store block length in
	60	B6	*LDA A*	location 005F.
	61	m_H		Move next character
	62	m_L		into *AC*.
	63	B7	*STA A*	Print character
	64	80		and clear flag f_7.
	65	04		
	66	7A	*DEC n*	Decrement the
	67	00		block length.
	68	5F		
	69	27	*BEQ L1*	If block length is zero,
	6A	04		jump to *L1*.
	6B	7A	*DEC* m_L	Decrement pointer
	6C	00		to next character.
	6D	62		
	6E	3B	*RTI*	Return.
L1:	6F	B7	*STA A*	Disable
	70	40		flag f_7.
	71	00		
	72	3B	*RTI*	Return.

5.9 REFERENCES

1. Zissos, D. and Duncan F. G. 'Digital Interface Design', Oxford University Press, 1973.
2. Zissos, D. 'Problems and Solutions in Logic Design', Oxford University Press, 1976.
3. INTEL 8080 Microprocessor Systems User's Manual, September, 1975.
4. M6800 Microprocessor Applications Manual, Motorola, 1975.

6

D.M.A. Systems

In this chapter we shall be concerned with the design and implementation of *direct memory access systems*, that is with systems that allow data to be transferred directly between a peripheral and the memory in a microprocessor system. The design philosophy and design steps are outlined in Chapter 2, sections 2.9 and 2.10 respectively.

6.1 INTRODUCTION

In the methods we have discussed so far for transferring data between a microprocessor and a peripheral (wait/go, test-and-go, and interrupts), the information moves through the m.p.u., as shown in Figure 6.1(a). These methods require several instructions to be executed for the transfer of each byte. For example, if we use the interrupt mode, we must

1. Disable further interrupts, if not automatically disabled
2. Store the re-entry point
3. Identify source of interruption
4. Service the request
5. Clear the flag
6. Restore the re-entry point
7. Enable interrupts, and
8. Return to the current program, as explained in chapter 5.

For large blocks of data this can involve excessive amounts of m.p.u. time. Furthermore, if the rate of the incoming information is greater than the rate at which the microprocessor can write into memory, some of the information will clearly be lost.

In a system, however, in which a direct link is established between memory

and peripheral, as shown diagrammatically in Figure 6.1(b), the above-mentioned problems would be eliminated, since we should be able to transfer a byte of information between memory and a peripheral in one memory cycle. Contrary to common belief, the design and implementation of such systems is straightforward. As we shall see later on in the chapter, the interface hardware is uncomplicated and the software required to drive it minimal, approximately a dozen instructions for each block transfer.

The duration of a direct memory access is normally expressed in *memory cycles. A memory cycle is a time cycle in which the appropriate signals to read or write into memory are generated.* Because a memory cycle is effectively 'stolen' from the m.p.u. operation each time a d.m.a. cycle is executed, a d.m.a. cycle is often referred to as *cycle steal.*

All microprocessors have a facility that allows the designer to establish a direct link between the microprocessor memory and a peripheral, as shown in Figures 6.1(b). This facility is called a *direct memory access*, or *d.m.a.* in abbreviated form.

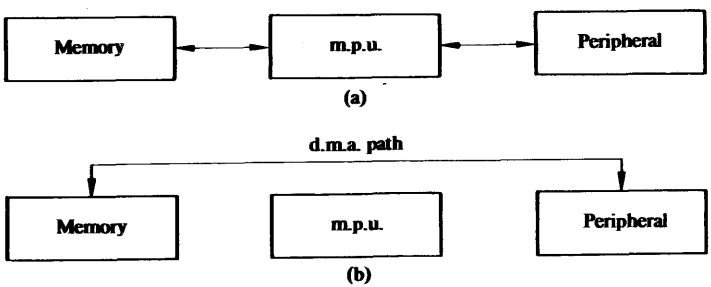


Figure 6.1.

6.2 D.M.A. SYSTEMS

The block diagram of our d.m.a. system is shown in Figure 6.2. It has been designed as a general system to accommodate any type of peripheral and all types of microprocessors. Its step-by-step operation is flowcharted in Figure 6.3 and is summarized below.

1. The programmer initializes the d.m.a. interface. He does so by using I/O instructions to output to the interface the initial memory address and the block length. The d.m.a. interface is, therefore, allocated an I/O address by means of which the programmer can access it.

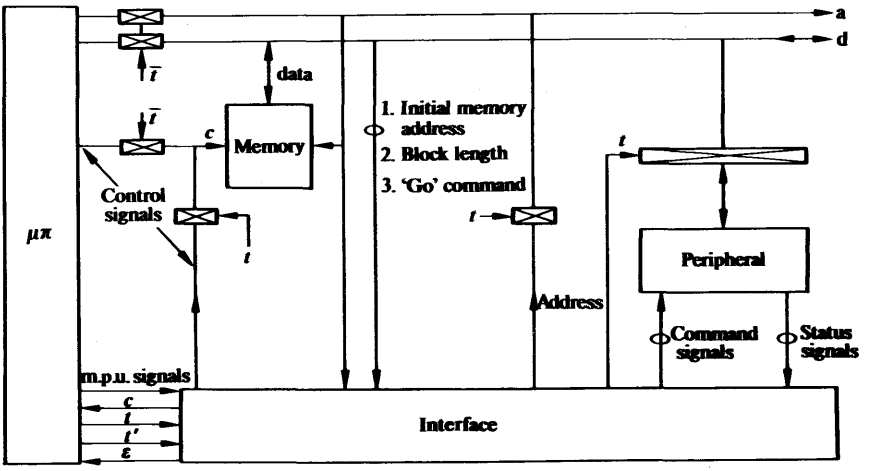


Figure 6.2.

2. He next outputs a third item to the interface, a 'go' command. This signal is used to initiate the block transfer, which is then completed autonomously, that is without programmer intervention.
3. When the block transfer is completed, the interface generates a flag signal, ε, to inform the programmer that the block transfer has been completed.
4. The programmer acknowledges flag ε by executing an I/O instruction, which clears it. Signals *c*, *t* and *t'* are defined in the next section.

6.3 D.M.A. INTERFACES

The design of d.m.a. systems, as already stated, is straightforward and should present no difficulty to the reader who possesses a working knowledge of logic design, outlined in Chapter 1.

A d.m.a. interface is best designed as two separate units, to which we shall refer as *interface 1* and *interface 2*—see Figure 6.4. The basic function of interface 1 is to accept the initial information output by the programmer and to generate the *start* and *stop* signals for interface 2, whose basic function is to control the transfer of data between the memory and the peripheral.

The step-by-step operation of each interface is described below.

Interface 1

Its block diagram is shown in Figure 6.5. It consists of two counters

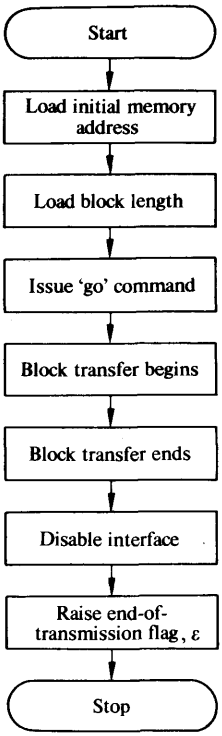


Figure 6.3.

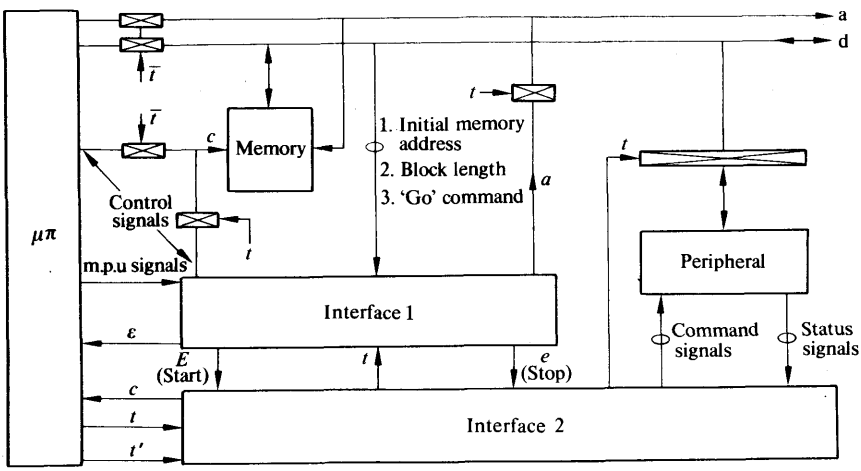


Figure 6.4.

connected in cascade, five gates and a flip-flop. The two counters are used to accept the initial memory address and the block length. They are loaded as follows. The programmer transfers into the accumulator the initial memory address and executes an I/O instruction with address A_p. This generates an I/O pulse on the load terminal of the two counters, which transfers the accumulator contents (the initial memory address) into counter 1. At the same time, because the two counters are connected in cascade, the contents of counter 1 are pushed into counter 2. The programmer then transfers into the accumulator the block length and executes the same I/O instruction. This causes the memory address in counter 1 to be pushed into counter 2, and the value of the block length (held in the accumulator) to be loaded into counter 1.

Next the programmer executes another I/O instruction with a different address, denoted in Figure 6.5 by A_q. This generates a pulse on the clock terminal of our flip-flop, shown as a *JKFF*. Because $J = 1$ the flip-flop, which has been initially reset by the system reset signal, sets turning signal E on. This signal, as we shall see later, causes interface 2 to start the block transfer, by initiating a d.m.a. cycle each time the peripheral becomes ready.

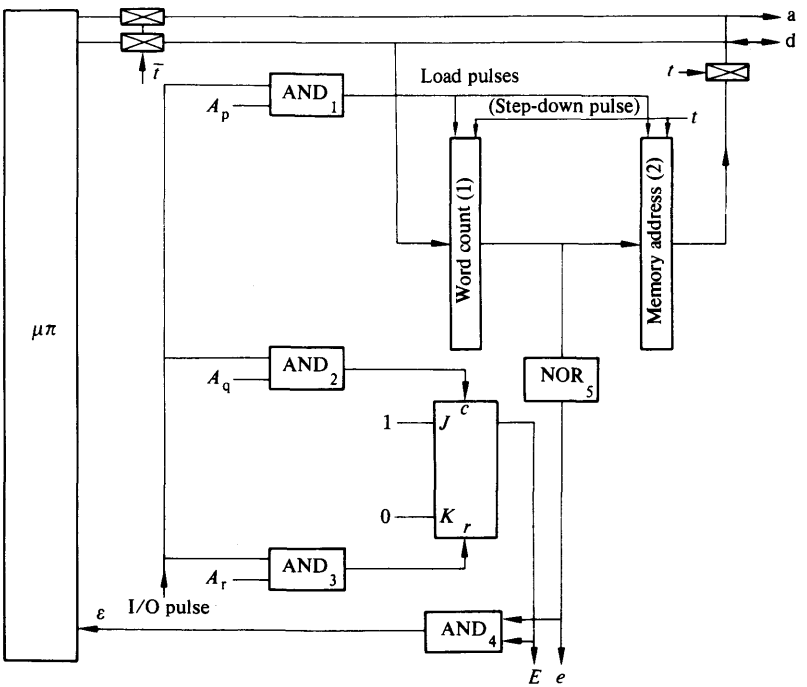


Figure 6.5.

While a d.m.a. cycle is being executed, signal t equals 1. We use this signal to decrement our two counters in Figure 6.5.

At the end of the block transfer counter 1 contains all zeroes, which is indicated by a logic 1 generated at the output of the NOR gate. This signal, denoted by variable e in Figure 6.5, is used by interface 2 to stop the block transfer. At the same time it is ANDed with flip-flop signal E to generate the end-of-block transfer signal, ε, which, as we have already explained, informs the programmer that the block transfer has been completed. The programmer acknowledges flag ε by executing an I/O instruction with address A_r. This, when decoded by AND gate 3 in Figure 6.5, generates a pulse on the flip-flop's reset terminal r, turning off signal ε.

In Figure 6.6 we show the block diagram of interface 1 for a microprocessor with an eight-line data bus and a sixteen line address bus. For the sake of clarity we do not show the peripheral in this diagram. Signals c, t and t' are explained next.

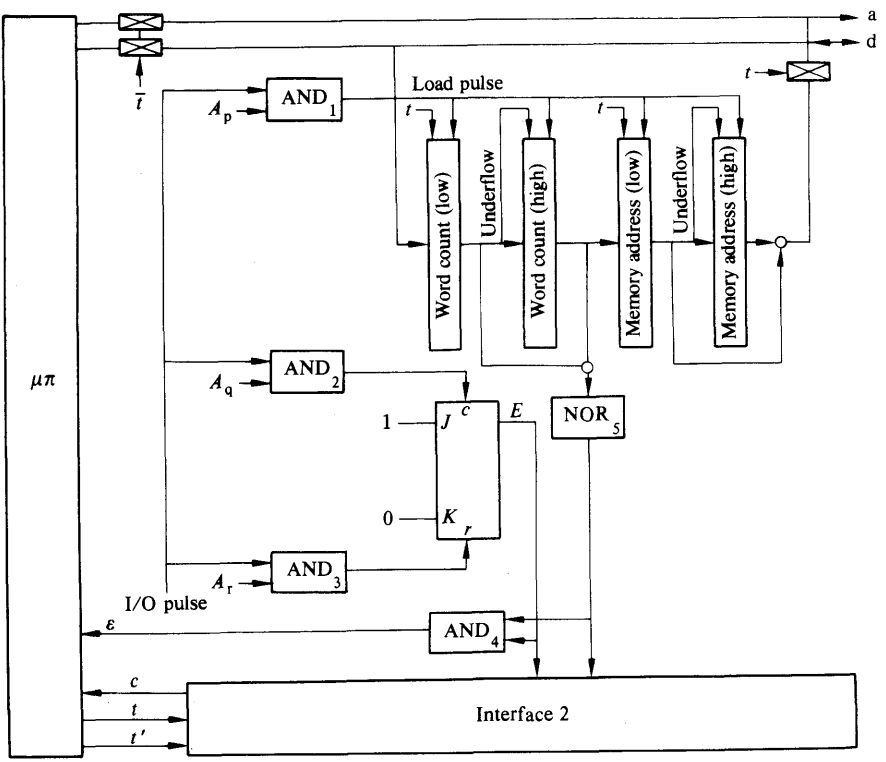


Figure 6.6.

Interface 2

The function of interface 2, as we have already stated, is to control the transfer of data between the microprocessor memory and the peripheral. To make our designs as far as possible microprocessor-independent, we shall assume the existence of a microprocessor with cycle-steal characteristics†, as described below.

Signal c. A 0 to 1 signal transition on this terminal tristates the microprocessor for one d.m.a. cycle. This, as we saw in section 6.1 is a time cycle during which the appropriate signals to read or write into memory are generated.

Signal t. This signal equals 1 when the microprocessor is tristated, otherwise $t = 0$. It is used to define the duration of a d.m.a. cycle.

Signal t'. Signal t' is generated during the presence of signal t, and is used as the read/write pulse for memory.

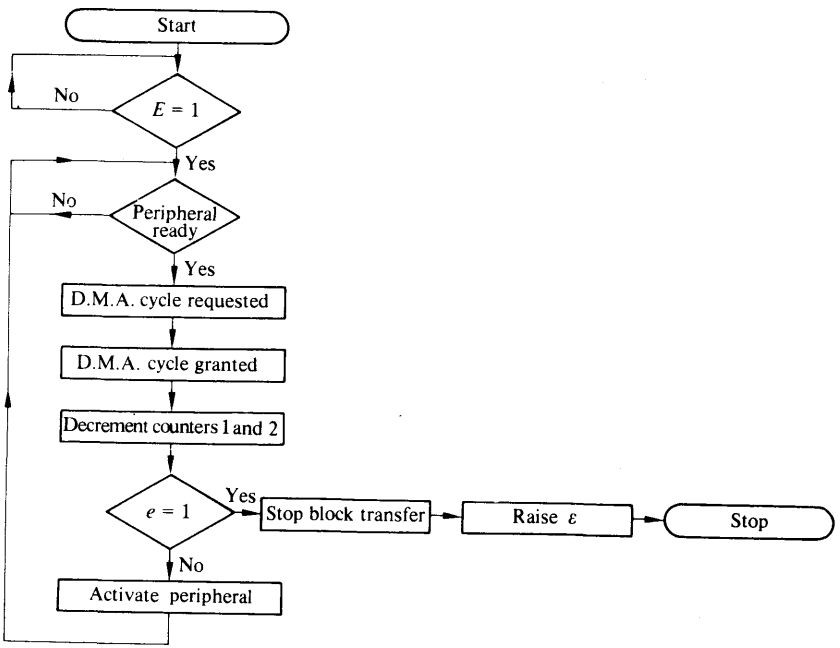


Figure 6.7.

† The implementation of microprocessor cycle-steal characteristics is discussed in section 6.5.

6.4 THE TWO-WIRE INTERFACE

In this section we show that the implementation of interface 2 in Figure 6.4 between a microprocessor with cycle-steal characteristics (explained in the previous section) and an action/status device† consists of two wires. To prove this result we shall use the design steps outlined in Chapter 1, section 9. For the sake of simplicity, initially we shall ignore the presence of signals E, e and t' in Figure 6.8. This does not invalidate our results.

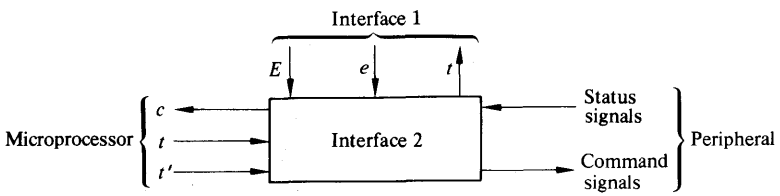


Figure 6.8.

Step 1 *I/O characteristics*

The I/O (input/output) signals are shown in Figure 6.9. The desired relationship between input and output is as described in the previous section, that is a d.m.a. cycle is requested by the peripheral when it becomes ready to read or write a byte in memory. When the d.m.a. cycle is granted, the transfer of one byte takes place, at the end of which the microprocessor resumes its activity and the peripheral is triggered into action. See Figure 6.10.

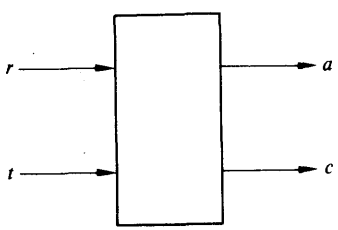


Figure 6.9.

Step 2 *Internal characteristics*

The internal state diagram of a suitable circuit is shown in Figure 6.11. Its operation is as follows. State S_0 is maintained while the microprocessor is tristated, that is during a d.m.a. cycle. At the end of the d.m.a. cycle, indicated by signal t changing to 0, the circuit moves to state S_1. The S_0 to S_1 transition

† Action/status devices are explained in Appendix 1.

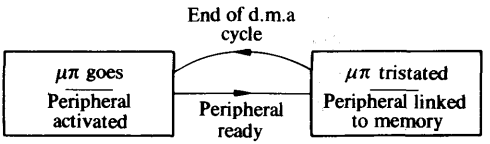


Figure 6.10.

causes action signal a to change from 0 to 1, triggering the peripheral into action. When the peripheral responds, indicated by its ready signal becoming 0, our circuit moves to state S_2. The circuit remains in state S_2 while the peripheral is responding, that is while r equals zero. When the device has fully responded, signal r changes to 1, which causes the circuit to move to state S_1. In moving from state S_2 to state S_1, signal c changes from 0 to 1 requesting a cycle steal from the microprocessor. When the cycle steal is granted, indicated by $t = 1$, our circuit moves to state S_0. The cycle repeats itself.

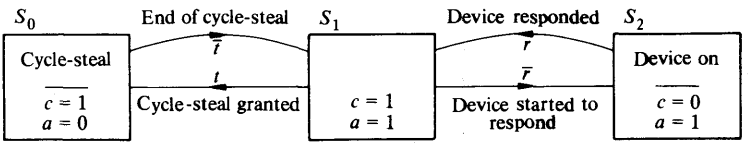


Figure 6.11.

Step 3 *State reduction*

In this step, as we have already explained, we translate the internal state diagram into a state table and apply Caldwell's reduction steps. The table derived directly from our state diagram in Figure 6.11 is shown in Figure 6.12(a). Applying the reduction steps (section 7, Chapter 1), allows its three rows to merge into one, shown in Figure 6.12 (b).

Step 4 *Circuit implementation*

By direct reference to the reduced state table, we obtain

$$c = \bar{t} \cdot r + t \cdot r + (\bar{t} \cdot \bar{r}) = r \tag{1}$$

$$a = \bar{t} \cdot \bar{r} + \bar{t} \cdot r + (\bar{t} \cdot \bar{r}) = \bar{t} \tag{2}$$

Introducing E and e, we obtain

$$c = E \cdot \bar{e} \cdot r \tag{3}$$

$$a = \bar{t} \tag{4}$$

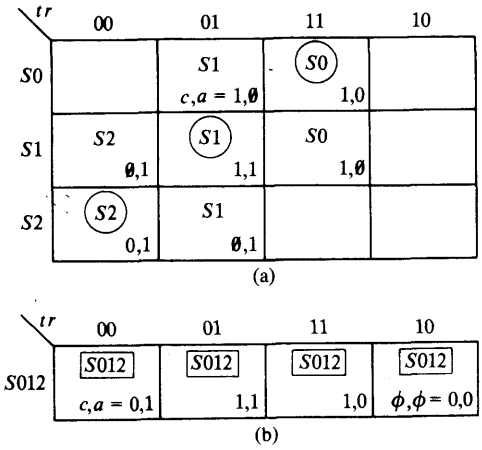


Figure 6.12.

That is, the d.m.a. interface between a cycle-steal microprocessor and action/status peripheral consists of two wires, as shown in Figure 6.13.

Reference to equations 6.3 and 6.4 indicates that the transfer of data starts with a d.m.a. cycle and ends with the activation of the peripheral.

6.5 CYCLE-STEAL LOGIC

The cycle-steal characteristics which we defined in the previous section are not available in current microprocessors. It is, therefore, left to the user to design a front-end logic to implement our cycle-steal characteristics in a given microprocessor. The block diagram of such a circuit is shown in Figure 6.14(a). The design procedure is straightforward. The main difficulty likely to be experienced by the reader is the correct interpretation of the m.p.u. pin functions from published data.

As we mentioned earlier, unless we specify otherwise, we shall assume that a d.m.a. cycle extends over three clock pulses, as shown in Figure 6.14(b). We shall demonstrate the procedure by designing the cycle-steal logic for the INTEL 8080.

Cycle-steal Logic for the INTEL 8080

The relevant m.p.u. signals are shown in Figure 6.14(a). A logic 1 on the hold line (pin 13) disconnects the address and data buses. The response signal is labelled *HLDA* (Hold Acknowledge), and appears on pin 21. This signal goes

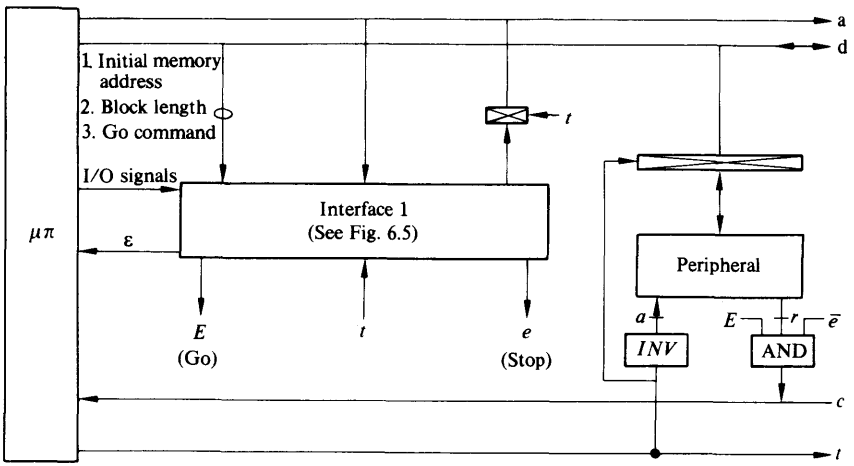


Figure 6.13.

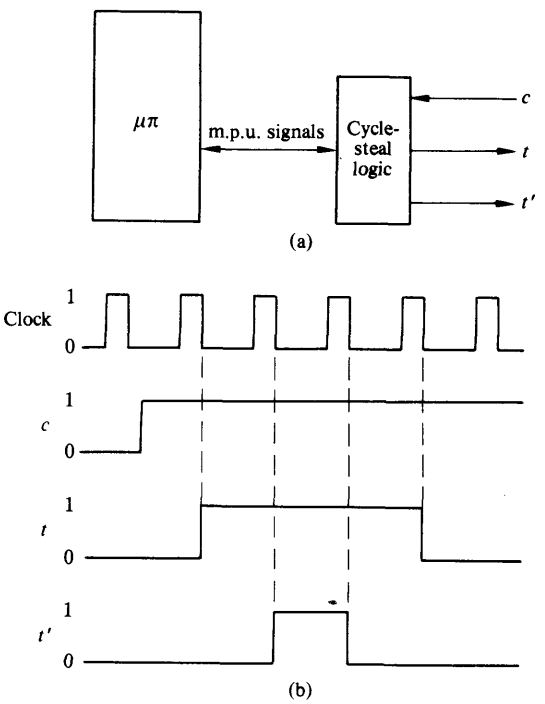


Figure 6.14.

high within a specified delay of the leading edge of ϕ_1 in machine state T3. The data and address buses are floated within a brief delay of the rising edge of the next ϕ_2 clock pulse. For a detailed timing diagram the reader is referred to the INTEL 8080 Microcomputer Systems User's Manual, September 1975, page 2–12.

In Figure 6.15(b) we show the internal state diagram of a logic circuit that tristates the INTEL 8080 for approximately three clock pulses.† By direct reference to this diagram, we obtain

$$S_A = S_1 = \bar{A}\cdot B, \qquad \text{therefore } J_A = B$$

$$R_A = S_3\cdot\bar{c}\cdot\overline{HLDA} = A\cdot\bar{B}\cdot\bar{c}\cdot\overline{HLDA}, \qquad \text{therefore } K_a = \bar{B}\cdot\bar{c}\cdot\overline{HLDA}$$

$$S_B = S_0\cdot HLDA = \bar{A}\cdot\bar{B}\cdot HLDA, \qquad \text{therefore } J_B = \bar{A}\cdot HLDA$$

$$R_B = S_2 = A\cdot B, \qquad \text{therefore } K_B = A$$

$$HOLD = S_0\cdot c + S_1 + S_2 = \bar{A}\cdot\bar{B}\cdot c + \bar{A}\cdot B + AB$$
$$= \bar{A}\cdot c + B$$

$$t = S_1 + S_2 + S_3\cdot HLDA = \bar{A}\cdot B + A\cdot B + A\cdot\bar{B}\cdot HLDA$$
$$= B + A\cdot\bar{B}\cdot HLDA$$
$$= B + A\cdot HLDA$$

$$t' = S_2 = A\cdot B$$

The corresponding (primitive) circuit is shown in Figure 6.16.

6.6 PROBLEMS AND SOLUTIONS

In this section we demonstrate our design steps by means of problems and fully-worked out solutions. The reader's attention is drawn to the fact that, although we use the INTEL 8080 to implement our designs, our procedures apply to all types of microprocessors with or without cycle-steal logic. Specifically, it should be noted that the first three steps in the design are executed without reference to the microprocessor.

D.M.A. Problem 1 *Reader to RAM interface*

Design a d.m.a. interface between an eight-channel paper tape reader and the RAM of a microprocessor system.

I/O addresses available are A_{004}, A_{005} and A_{006}.

† In this implementation signal c must not be removed before *HLDA* is generated.

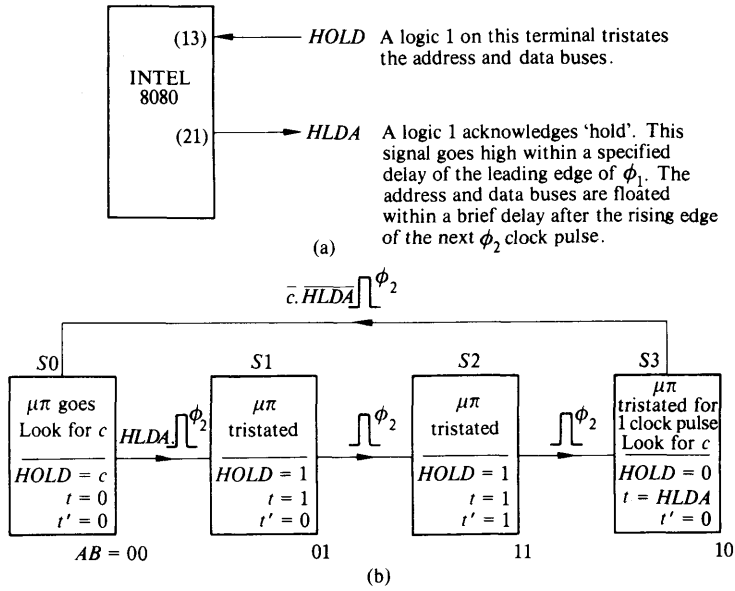


Figure 6.15.

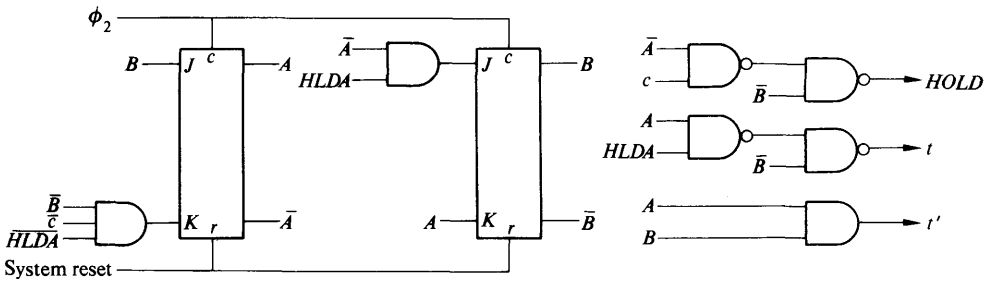


Figure 6.16.

SOLUTION

Step 1 *Aim of the design*

The aim of the design is to enable the programmer to transfer a block of characters from a data source directly into a RAM of a microprocessor system using the d.m.a. facility.

Step 2 *Device characteristics*

The microprocessor is tristated for three clock pulses on the leading edge of

a pulse on terminal *c*, as shown in Figure 6.14. Assuming a clock frequency of 0.5 to 1.5 MHZ, the duration of signal *t* is between 2 and 6 microseconds, and of *t'* between 667 nanoseconds and 2 microseconds.

The write cycle of the RAM to be used is shown in Figure 6.17.

The reader is an action/status device—see Appendix 1.

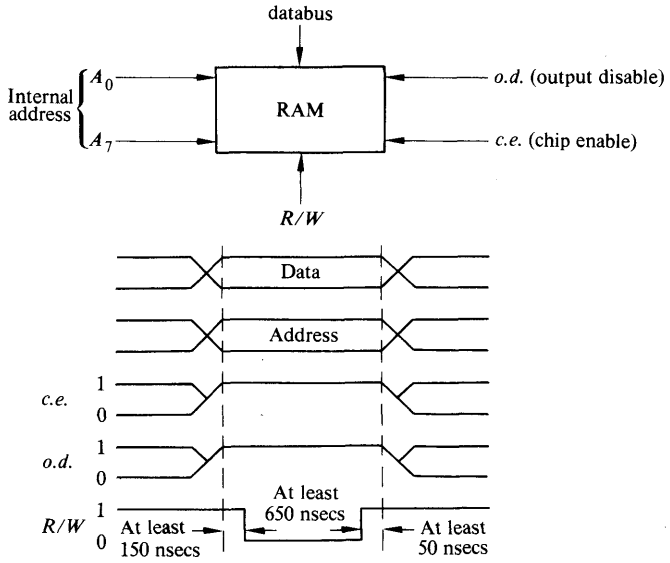


Figure 6.17.

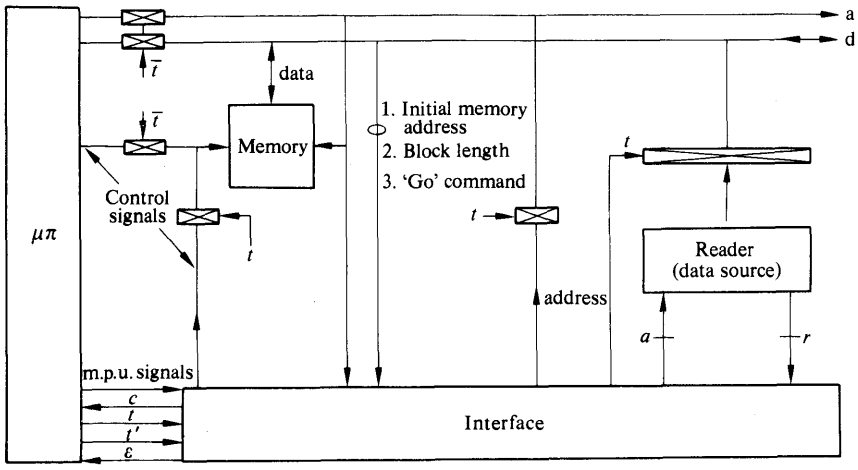


Figure 6.18.

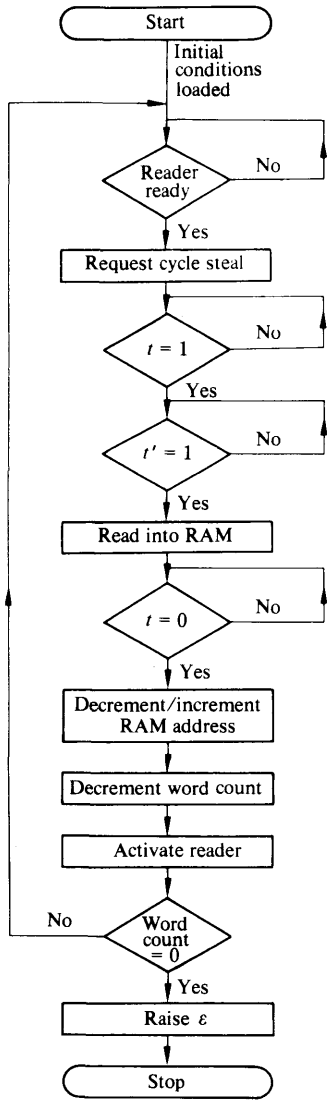

Start
Initial conditions loaded
Reader ready
No
Yes
Request cycle steal
t = 1
No
Yes
t' = 1
No
Yes
Read into RAM
t = 0
No
Yes
Decrement/increment RAM address
Decrement word count
Activate reader
No
Word count = 0
Yes
Raise ε
Stop

Figure 6.19.

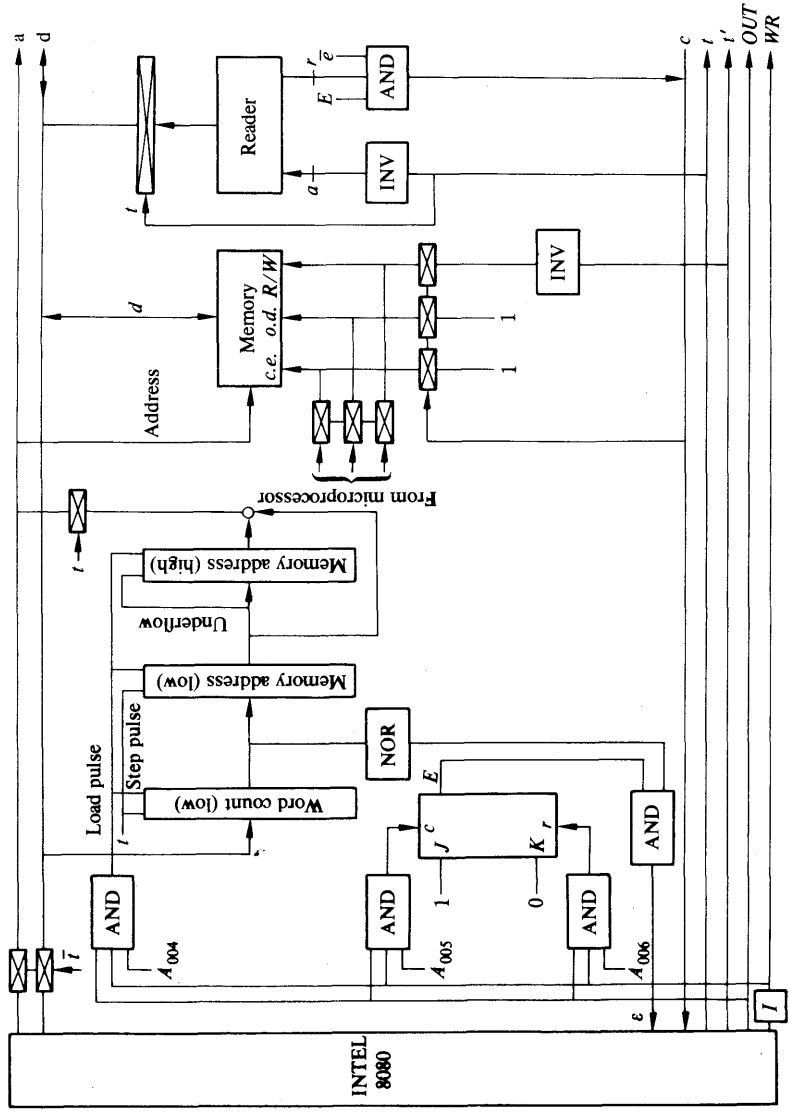

a
d
c
t
t'
OUT
WR
Reader
AND
INV
Memory
c.e. o.d. R/W
Address
From microprocessor
Memory address (high)
Underflow
Memory address (low)
Step pulse
Load pulse
Word count (low)
NOR
AND
INV
A004
A005
A006
INTEL 8080

Figure 6.20.

Step 3 *System design*

The block diagram of our solution is shown in Figure 6.18. Its step-by-step operation, after the initial conditions have been loaded, is flow-charted in Figure 6.19.

Step 4 *Hardware design*

Our starting point is the block diagram shown in Figure 6.4, showing a general d.m.a. system. The implementation of interfaces 1 and 2 is shown in Figure 6.5 and 6.13. If we assume that the block length is not greater than 256 words, then the corresponding system using the INTEL 8080 is shown in Figure 6.20.

Step 5 *Software design*

The software required to initiate the transfer of n characters from the reader into consecutive locations in RAM, given the interface in Figure 6.20, is

Load *AC* with initial RAM address (high)	Load initial RAM address.
OUT	
004	
Load *AC* with initial RAM address (low)	
OUT	
004	
Load *AC* with block length	Load block length.
OUT	
004	
OUT	Start block transfer.
005	
When $\varepsilon = 1$ execute service routine	
OUT	Clear flag.
006	
EI	Enable interrupts.
RET	Return.

D.M.A. problem 2 *System modification*

Modify the solution of the previous problem so that only valid data, indicated by $V = 1$, are read into RAM.

SOLUTION

The specified modification is implemented by suppressing a d.m.a. cycle and initiating a read cycle when $\bar{V} \cdot r = 1$ [Introducing variable r in the equation

ensures that the data is sampled only when $r = 1]$. This is achieved by modifying signals c and a in Figure 6.20 to the following form

$$c = Er\bar{e}V$$

$$a = t\overline{V}r.$$

7
D.D.T. Systems

In this chapter we shall be concerned with the design and implementation of direct data transfer (d.d.t.) microprocessor systems, that is with microprocessor systems that allow data to be transferred directly between peripherals. The design philosophy and design steps are outlined in Chapter 2, sections 9 and 10, respectively.

7.1 INTRODUCTION

Let us consider the following situation. We wish to obtain a print-out of some information which is punched on a tape. We are told that for this purpose we can use a tape reader and a printer in a microprocessor system with one condition. We must not tie up the microprocessor for more than a few microseconds each time we produce a print-out.

The methods available to us are

1. To read each character into the accumulator and print it, as shown in Figure 7.1. This method requires several instructions to be executed for the transfer of each byte from the reader to the printer, as the solution of problem 3 in Chapter 4 shows. As the execution time of each instruction is typically a few microseconds, this method is automatically eliminated.
2. To read the complete tape into memory (RAM) and then print it, using a d.m.a. channel in each case, as shown in Figure 7.2. If there are n characters on the tape, we shall need $2 \times n$ d.m.a. cycles for the actual data transfer plus half a dozen or so instructions to initialize the two interfaces, as we explained in the previous chapter. The total time involved in this case, although considerably less than the time involved in the previous method, is still likely to be over 100 microseconds with present-day microprocessors.

This figure is well in excess of the 'few microseconds' specified in the problem. Furthermore, this method assumes that memory (RAM) space is available for this purpose, which may not always be the case, and requires 2 d.m.a. channels.

3. To establish a direct data link between the reader and the printer, as we show diagrammatically in Figure 7.3. In such a system the data is transferred between the source and the acceptor independently of the microprocessor, which is left free to continue with its own activities, thus allowing us to meet the imposed restriction.

For the sake of clarity, in Figures 7.1, 7.2 and 7.3 we do not show the other peripherals.

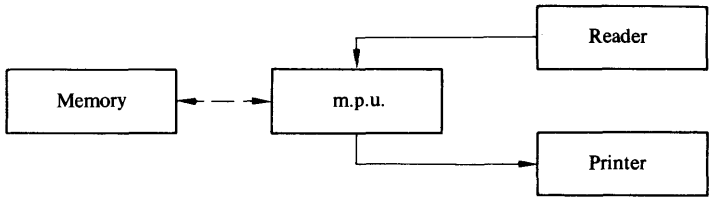


Figure 7.1.

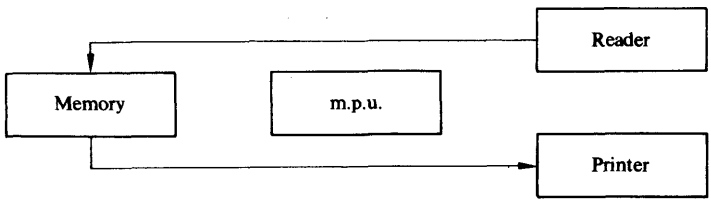


Figure 7.2.

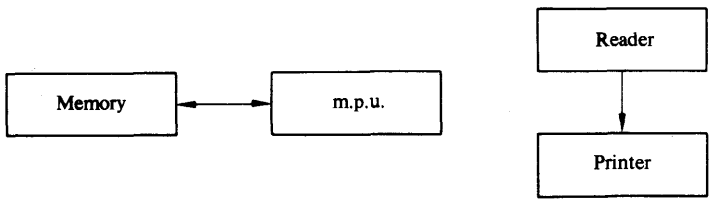


Figure 7.3.

7.2 D.D.T. SYSTEMS

The block diagram of our d.d.t. system is shown in Figure 7.4. The interface hardware is given an I/O (input/output) address by means of which it can be accessed. When the user wishes to establish a direct data link between a pair (or a set) of peripherals, he proceeds as follows.

He first determines whether the peripherals he intends to use are available or not. He does so by polling signal b in Figure 7.4. This is a status signal that indicates whether the peripherals in question are busy or not; b equals 1 when one or more of the peripherals are *busy*, otherwise $b = 0$. If the devices are found to be *free* (not busy) he executes an I/O instruction with the address given to the interface. The interface responds by (i) isolating from the system the section of the bus that links the peripherals in question, (ii) changing the value of the busy signal b from 0 to 1, and (iii) initiating the data transfer which then takes place autonomously, that is without programmer intervention. When the complete data block has been transferred, signal b changes to 0 and the bus section becomes untristated.

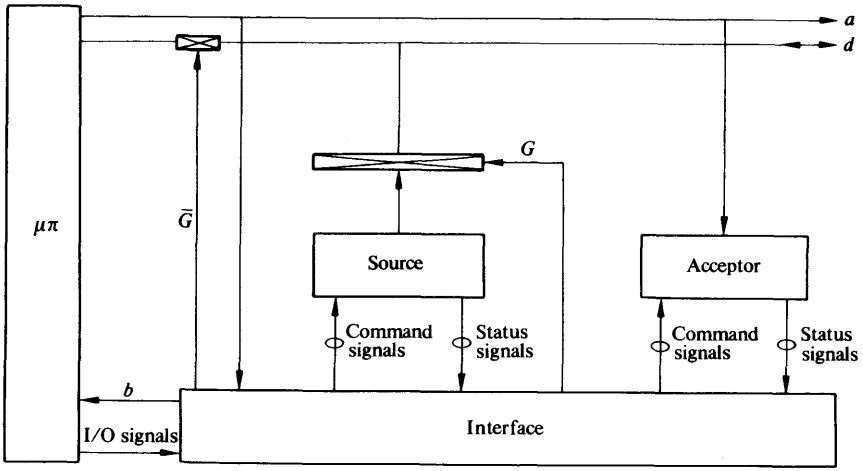


Figure 7.4.

7.3 D.D.T. INTERFACES

The design and implementation of d.d.t. interfaces follow well-established procedures and should present no difficulty to the reader who possesses a working knowledge of logic design.[1]

In the case of action/status devices,† it has been shown that the interface hardware consists of two wires.[2] For the sake of completeness we reproduce the proof below.

Our starting point is the block diagram in Figure 7.5(a). Its operation is flow-charted in Figure 7.5(b). For clarity of design we show no external control signals at this stage.

From the implementation point of view, the interface is a logic circuit with r_1 and r_2 as input signals, and a_1 and a_2 as output signals, see Figure 7.6. We shall use the steps outlined in Chapter 1, section 9 to implement our circuit, as we show below.

Step 1 *External characteristics*

See Figure 7.5.

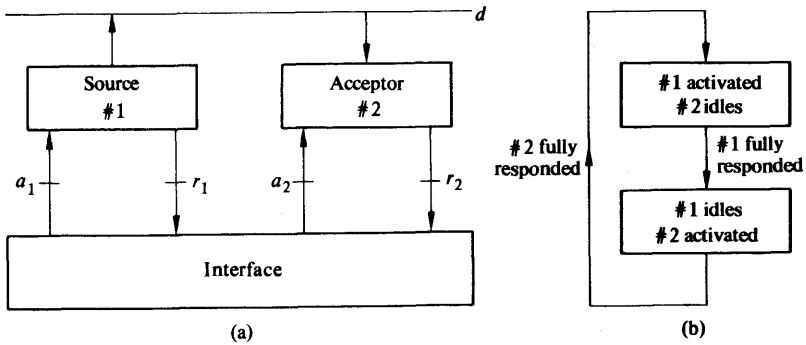


Figure 7.5.

Step 2 *Internal characteristics*

A suitable internal state diagram is shown in Figure 7.7. Its operation is as follows.

Let us assume that device 1 is active and device 2 inactive. The corresponding state in our diagram is S_0. This state is maintained while device 1 remains active. When it has fully responded, indicated by r_1 changing to 1, our circuit moves to state S_1. Reference to Figure 7.7 shows that a_2 equals 0 in state S_0 and 1 in state S_1, that is the S_0 to S_1 circuit transition causes action signal a_2 to change from 0 to 1. This signal change activates device 2. When r_2 changes to 0, indicating that device 2 has begun to respond, our circuit moves to state S_2. It remains in this state until device 2 has fully responded, indicated by r_2 changing to 1. When r_2 equals 1 our circuit moves to state S_1. As a_1 equals 0 in

† See Appendix 1.

state S_2 and 1 in state S_1, the S_2 to S_1 circuit transition activates device 1. When it responds, indicated by signal r_1 changing to 0, our circuit moves to state S_0, where it remains until device 1 has fully responded. The cycle repeats itself.

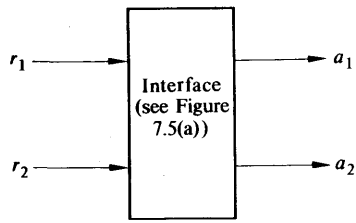


Figure 7.6.

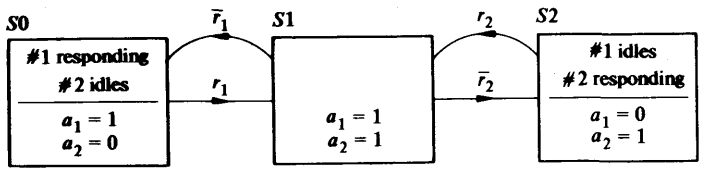


Figure 7.7.

Step 3 *State reduction*

The state table corresponding to our state diagram in Figure 7.7 is shown in Figure 7.8(a). Applying the reduction steps described in section 7 of Chapter 1, the three rows of the table merge into a single row, as shown in Figure 7.8(b). In the first square we enter circled entry S_{012}, since there is no other state that our circuit can assume. The outputs in this square at this stage are ϕ, ϕ, indicating optional values.

Step 4 *Circuit implementation*

By direct reference to the reduced state table in Figure 7.8(b), we obtain

$$a_1 = \bar{r}_1 \cdot r_2 + r_1 \cdot r_2 + (\bar{r}_1 \cdot \bar{r}_2) = r_2 \qquad 7(1)$$

$$a_2 = r_1 \cdot r_2 + r_1 \cdot \bar{r}_2 + (\bar{r}_1 \cdot \bar{r}_2) = r_1 \qquad 7(2)$$

Since the optional product $\bar{r}_1 \cdot \bar{r}_2$ has not been used in the derivation of our final expressions for signals a_1 and a_2, $a_1 = a_2 = 0$ in the first square in Figure 7.8(b).

We shall refer to equations 7.1 and 7.2 as *primitive interface equations*. Their implementation consists of two wires, as shown in Figure 7.9.

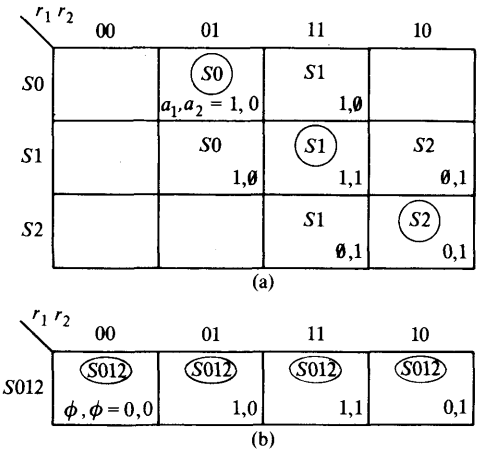


Figure 7.8.

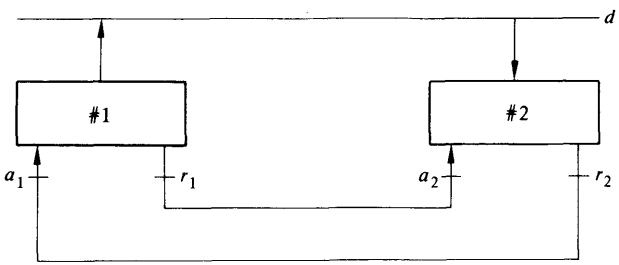


Figure 7.9.

Go/No-go Control

In order that we do not lose information, unless we specify otherwise, we shall start a data transmission with a read operation and end it with a write operation, as shown in Figure 7.10. Our read and write operations are as defined in Figure 7.11. We use signal G to define the active ($G = 1$) and the inactive ($G = 0$) states of our system—see Figure 7.12. By direct reference to this diagram, we obtain

$$a_1 = \bar{G} \cdot r_2 + G \cdot r_2 = r_2$$

$$a_2 = G \cdot r_1$$

The corresponding circuit is shown in Figure 7.13.

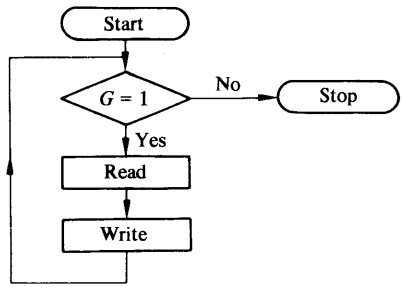


Figure 7.10.

Figure 7.11.

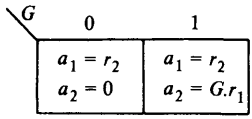


Figure 7.12.

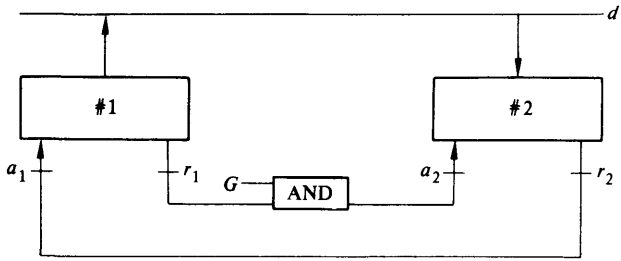


Figure 7.13.

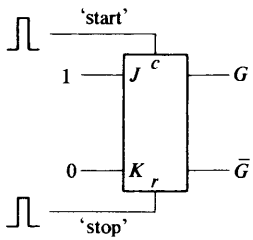


Figure 7.14.

If the start/stop commands are pulses, we can use the JK flip-flop in Figure 7.14 to generate control signal G. Other implementations are clearly possible.

Read Inhibit ($i_2 = 1$)

In our system a read inhibit operation is implemented by suppressing a read operation ($a_2 = 0$) and initiating in its place a write operation ($a_1 = G \cdot r_1$), as shown in Figure 7.15. By direct reference to this Figure, we obtain

$$a_1 = r_2 \cdot \bar{i}_2 + G \cdot r_1 \cdot i_2$$

$$a_2 = G \cdot r_1 \cdot \bar{i}_2$$

i_2	0	1
	$a_1 = r_2$ $a_2 = G.r_1$	$a_1 = G.r_1$ $a_2 = 0$

Figure 7.15.

Write Inhibit ($i_1 = 1$)

As in the case of the read inhibit, we implement a write inhibit cycle by suppressing a write operation ($a_1 = 0$) and initiating instead a read operation ($a_2 = G \cdot r_2$), as shown in Figure 7.16. By direct reference to this Figure, we obtain

$$a_1 = r_2 \cdot \bar{i}_1$$

$$a_2 = G \cdot r_1 \cdot \bar{i}_1 + G \cdot r_2 \cdot i_1$$

i_1	0	1
	$a_1 = r_2$ $a_2 = G.r_1$	$a_1 = 0$ $a_2 = G.r_2$

Figure 7.16.

$i_1 i_2$	00	01	11	10
	$a_1 = r_2$ $a_2 = G.r_1$	$a_1 = G.r_1$ $a_2 = 0$	$a_1 = 0$ $a_2 = 0$	$a_1 = 0$ $a_2 = G.r_2$

Figure 7.17.

Interface Equations

The values of the action signals for all combinations of read and write inhibit signals are displayed in Figure 7.17. By direct reference to it, we obtain

$$a_1 = \bar{i}_1 \cdot \bar{i}_2 \cdot r_2 + \bar{i}_1 \cdot i_2 \cdot G \cdot r_1 \qquad 7(3)$$

$$a_2 = i_1 \cdot \bar{i}_2 \cdot G \cdot r_1 + i_1 \cdot \bar{i}_2 \cdot G \cdot r_2 \qquad 7(4)$$

We shall refer to the above equations as our *interface equations.*

7.4 PROBLEMS AND SOLUTIONS

In this section we demonstrate our design steps by means of problems and fully-worked out solutions. The reader's attention is drawn to the fact that although we use the INTEL 8080 to implement our designs, our procedures apply to all types of microprocessors. Specifically, it should be noted that the first three steps in the design are executed without reference to the microprocessor.

Problem 1 *Copy a tape*

Reproduce a given paper tape using the d.d.t. mode.

SOLUTION

Step 1 *Aim of the design*

The aim of the design is to move data from a source to an acceptor using the d.d.t. mode. No processing of data is required.

Step 2 *Device characteristics*

The reader and punch are action/status devices. In addition the reader generates an end-of-tape (e.o.t.) signal.

Step 3 *System design*

The block diagram of our solution is shown in Figure 7.18. Its step-by-step operation is flow-charted in Figure 7.19.

Step 4 *Hardware design*

Reference to Figure 7.18 shows that the signals to be generated by our interface are: a_1, a_2, b and G.

Signals a_1 and a_2 are derived directly from our interface equations

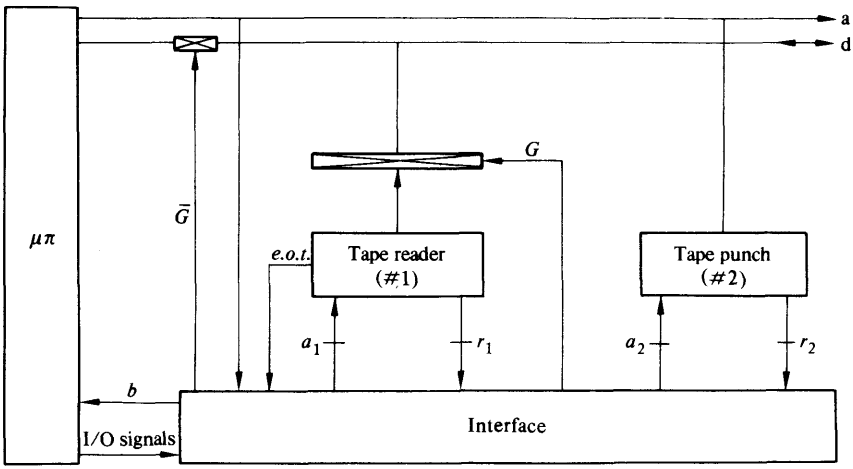


Figure 7.18.

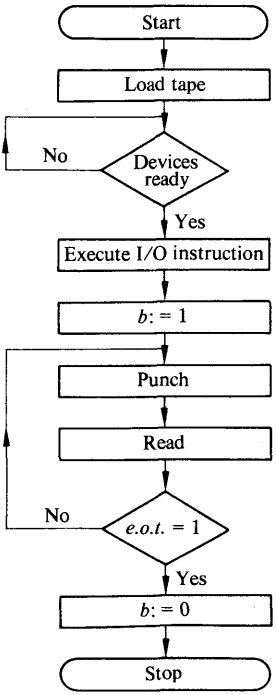


Figure 7.19.

(equations 7.3 and 7.4) by substituting 0s for i_1 and i_2.

$$a_1 = r_2$$

$$a_2 = G \cdot r_1$$

The equation for signal b is

$$b = \bar{r}_1 + \bar{r}_2 + e.o.t.\text{—see Figure 7.20}$$

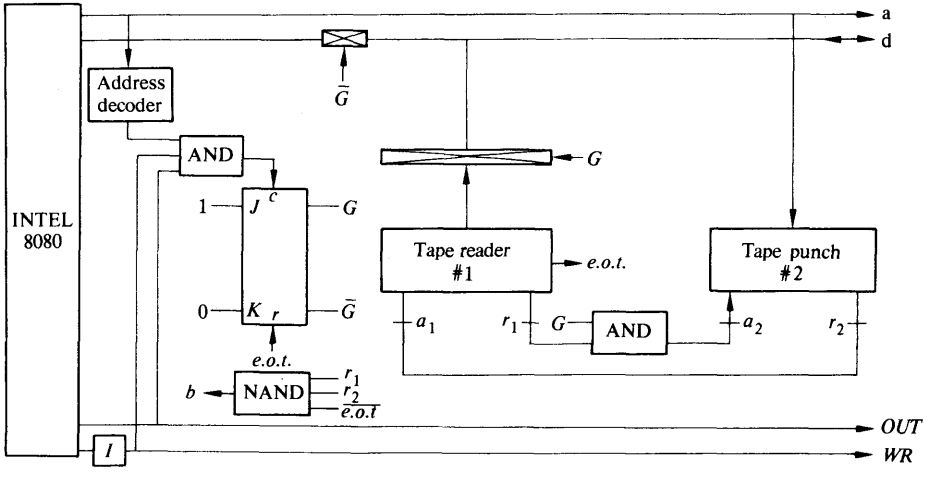


Figure 7.20.

We use a flip-flop, in our case a *JKFF*, to generate the go/no-go signal G. We set the flip-flop with an I/O instruction. It resets automatically with the end-of-tape (e.o.t.) signal.

Step 5 *Software design*

With the exception of the I/O 'go' instruction, no other software is required.

Problem 2 *Clean a tape*

Copy a paper tape deleting rub-out characters (all 1s).

SOLUTION

Step 1 *Aim of the design*

The aim of the design is to reproduce incoming data selectively. In our case rub-out characters (all 1s) will not be reproduced.

Step 2 *Device characteristics*

The tape reader and tape punch are action/status devices. In addition to the action/status signals, the tape reader generates an end-of-tape (e.o.t.) signal.

Step 3 *System design*

The block diagram of our solution is shown in Figure 7.21. Its step-by-step operation is flow-charted in Figure 7.22.

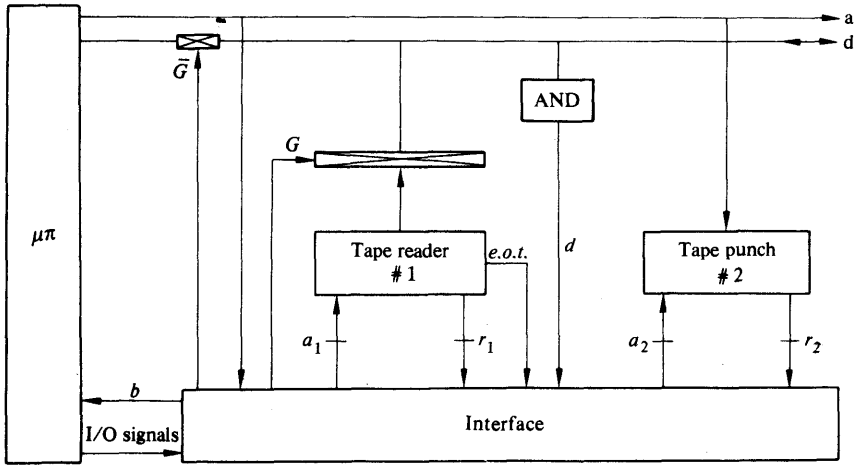


Figure 7.21.

Step 4 *Hardware design*

Reference to Figure 7.21 shows that our interface, as in the previous problem, is required to generate four signals, namely a_1, a_2, b and G.

Signals a_1 and a_2 are derived directly from our interface equations (equations 7.3 and 7.4) by substituting d and 0 for i_1 and i_2 respectively. Signal d' is generated by ANDing the eight data bus signals; that is $d = 1$ indicates a rub-out (delete) character. It is sampled when $r_1 = 1$, that is when the data from the reader is stable.

$$a_1 = \bar{d} \cdot r_1 \cdot r_2 + G \cdot r_1 \cdot d$$

$$a_2 = \bar{d} \cdot G \cdot r_1$$

The equation for signal b is

$$b = \bar{r}_1 + \bar{r}_2 + e.o.t.$$—see Figure 7.23.

Signal G, as in problem 1, is generated by the JK flip-flop in Figure 7.23. It is

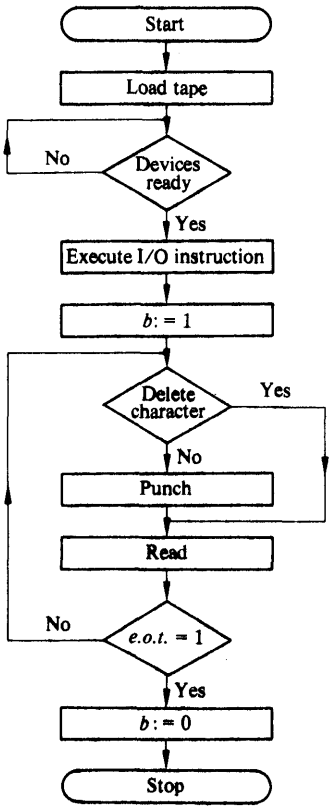


Figure 7.22.

set by the I/O 'go' instruction and is reset automatically with the end-of-tape (*e.o.t.*) signal.

Step 5 *Software design*

With the exception of the I/O 'go' instruction, no other software is required.

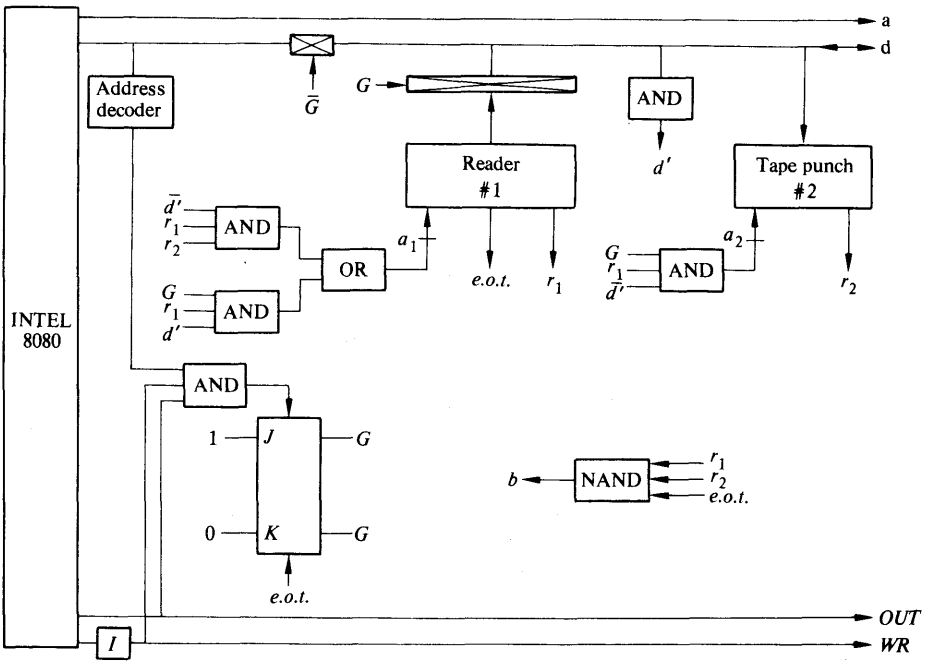


Figure 7.23.

7.5 REFERENCES

1. Zissos D. 'Problems and Solutions in Logic Design', Oxford University Press, 1976.
2. Zissos D., Duncan F. G. and Colin T. J. 'Logic-free Data Channels', Electronics Letters, Vol. 10, No. 17, August 22, 1974.

Appendix 1

Action/Status Devices

The concept of action/status devices is explained and a method for their implementation is outlined.

A1.1 ACTION/STATUS DEVICES

In 1974 Zissos, Duncan and Collin proved that the interface between a pair of action/status devices consists of two wires.[1, 2] Action/status devices have two terminals, *an action terminal* and *a status terminal*, as shown in Figure A1.1. Signals a and r have the following meaning.

Signal a. A 0 to 1 signal transition on the action terminal activates the device. No activation is possible when $r = 0$.

Signal r. This signal indicates the availability ($r = 1$) or unavailability ($r = 0$) of the device.

Now, most devices in practice require a sequence of command signals to operate them, and do not therefore fit our action/status model. Such devices, however, can be readily modified into action/status devices by means of some simple circuitry, the front-end logic, which we describe below.

A1.2 FRONT-END LOGIC

The block diagram of a front-end logic is shown in Figure A1.2. Its function is to monitor the status signals of the device and to generate the correct sequence of command signals to drive the device when action signal changes from 0 to 1. In addition it generates the status signal r. Its implementation is straightforward and uses the procedures outlined in Chapter 1.

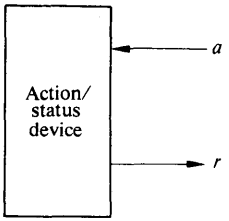


Figure A1-1.

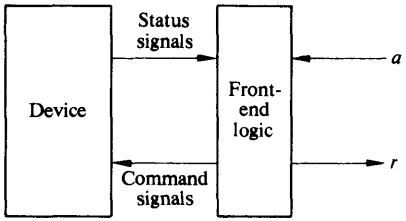


Figure A1-2.

The main difficulty likely to be experienced by the reader is the correct interpretation of the status and command signals from data supplied by the manufacturer. Such an exercise, although tedious, has been found in practice useful, in that it provides the user with the opportunity to clearly understand the operational features and idiosyncrasies of the device before he uses it. We give below some general guidelines for the design of front-end logic circuits.

In order that the input drive requirements of all devices equipped with front-end logic are identical, the action signal is not used to drive the device directly, but instead it is used to initiate the circuit transition to the next state. In this state we generate the first command signal and monitor the corresponding status signals. When the device has fully responded to the first command signal, our front-end logic generates the second command signal. The process continues until the device has fully responded to the last command signal. At this point the circuit assumes its initial state when $a = 0$. This ensures that the front-end logic responds only to the leading edge of an action pulse, and allows the device to free-run by connecting its ready (status) signal to its action terminal. We shall demonstrate our steps by means of the following example.

Example

Design the front-end logic for a digital printer (Figure A1.3), whose terminal characteristics are.

Terminal w A ground on this terminal ($w = 0$) positions the print wheels according to the input data.

Terminal x While the print wheels are being positioned, $x = 0$. This signal changes to 1 when the wheels are correctly positioned.

Terminal y Grounding terminal y causes the print hammers to strike and the paper to advance to its next line position.

Terminal z Signal $z = 0$ when the print hammers are being activated and the paper is advancing, otherwise $z = 1$.

In this example a, x and z are input signals and w, y and r are outputs. The state diagram of a suitable circuit is shown in Figure A1.4. By direct reference

to this diagram we obtain

turn on set of $A = B \cdot \bar{x}$
turn off set of $A = \bar{B} \cdot \bar{a} \cdot z \xrightarrow{invert} B + a + \bar{z}$
turn on set of $B = \bar{A} \cdot a$
turn off set of $B = A \cdot \bar{z} \xrightarrow{invert} \bar{A} + z$

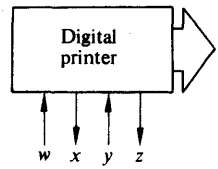


Figure A1-3.

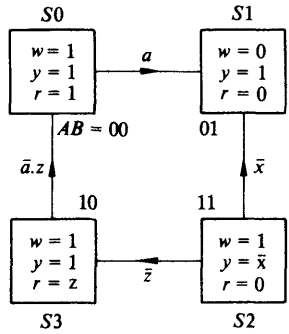


Figure A1-4.

Therefore, the circuit equations are

$$A = B \cdot \bar{x} + A \cdot (B + a + \bar{z})$$

$$B = \bar{A} \cdot a + B \cdot (\bar{A} + z)$$

$$w = \bar{S}_1 = \overline{\bar{A} \cdot B} = A + \bar{B}$$

$$y = S_0 + S_1 + S_2 \cdot \bar{x} + S_3$$

$$= \bar{S}_2 + S_2 \cdot \bar{x}$$

$$= \bar{S}_2 + \bar{x}$$

$$= \bar{A} + \bar{B} + \bar{x}$$

The circuit implementation of these equations shown in Figure A1.5 consistutes the front-end logic of the printer (Figure A1.5).

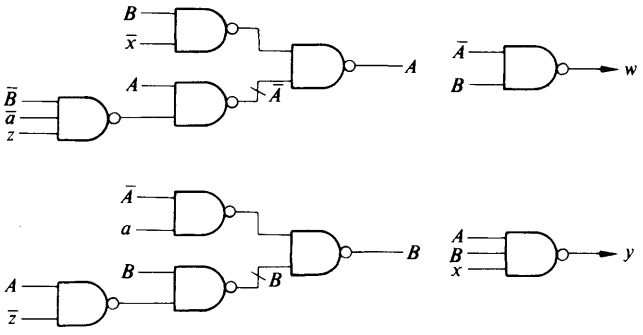


Figure A1-5.

A1.3 REFERENCES

1. Zissos, D., Duncan, F. G. and Collin, T. J. 'Logic-free Data Channels', Electronics Letters, Vol. 10, No. 17, August 1974.
2. Collin, T. J. 'Logic-free Data Channels', M.Sc. Thesis, University of Calgary, 1974.

Appendix 2

The INTEL 8085

A2.1 GENERAL [†]

The INTEL 8085 has been evolved from the INTEL 8080 as a result of evolutionary advances in technology, which have been taking place in the last ten years or so, and of the experience acquired using microprocessor systems. Its main features from the system designer's point of view are summarized below.

(i) *Single +5V Power Supply.* This feature is particularly useful in the case of portable equipment, as well as keeping the cost and complexity of small systems low.

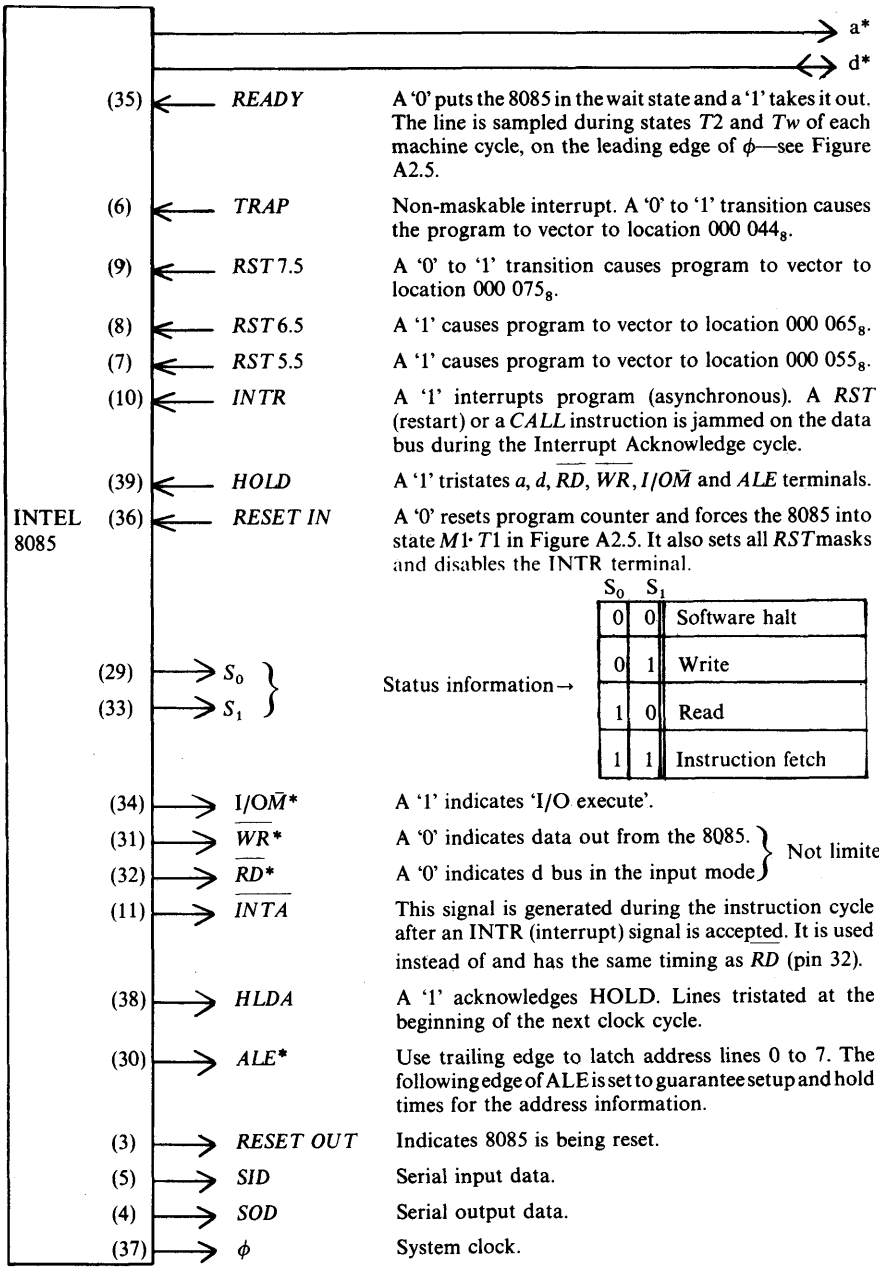
(ii) *Single System Clock.* The use of a single-phase clock reduces considerably the timing constraints on the interface signals that must be observed by the system designer. The minimum and maximum clock frequencies are 0.5 MHZ and 3 MHZ. The clock circuitry is built on the chip and only an external crystal or an RC network is required. The clock signal, ϕ, is output on pin 37—see Figure A2.1. The internal circuit transitions take place on the trailing edge of ϕ.

The reader's attention is drawn to the fact that internally a 50% duty cycle, two-phase, non-overlapping lock is generated from the external oscillator. One phase of this clock is made available to the user.

(iii) *Reduced Chip Count.* The high level of component integration allows a minimal system to be produced using three i.c. chips, namely 8085 (m.p.u.), 8155 (RAM) and 8355/8755 (ROM/PROM).

(iv) *M.P.U. Signals.* These are shown in Figure A2.1. Generally speaking they are clearly-defined and well-chosen, with perhaps two exceptions. (a)

†For a more detailed description of the INTEL 8085 see reference [1].



A '0' puts the 8085 in the wait state and a '1' takes it out. The line is sampled during states *T*2 and *Tw* of each machine cycle, on the leading edge of ϕ—see Figure A2.5.

Non-maskable interrupt. A '0' to '1' transition causes the program to vector to location 000 044_8.

A '0' to '1' transition causes program to vector to location 000 075_8.

A '1' causes program to vector to location 000 065_8.

A '1' causes program to vector to location 000 055_8.

A '1' interrupts program (asynchronous). A *RST* (restart) or a *CALL* instruction is jammed on the data bus during the Interrupt Acknowledge cycle.

A '1' tristates *a*, *d*, $\overline{RD}$, $\overline{WR}$, *I*/$O\bar{M}$ and *ALE* terminals.

A '0' resets program counter and forces the 8085 into state *M*1·*T*1 in Figure A2.5. It also sets all *RST* masks and disables the INTR terminal.

Status information →

S_0	S_1	
0	0	Software halt
0	1	Write
1	0	Read
1	1	Instruction fetch

A '1' indicates 'I/O execute'.

A '0' indicates data out from the 8085.
A '0' indicates d bus in the input mode
} Not limited to I/Os.

This signal is generated during the instruction cycle after an INTR (interrupt) signal is accepted. It is used instead of and has the same timing as *RD* (pin 32).

A '1' acknowledges HOLD. Lines tristated at the beginning of the next clock cycle.

Use trailing edge to latch address lines 0 to 7. The following edge of ALE is set to guarantee setup and hold times for the address information.

Indicates 8085 is being reset.

Serial input data.

Serial output data.

System clock.

Figure A2.1.

*Tristated during 'software halt'.

Terminal $I/O\bar{M}$ (pin 34 in Figure A2.1) is tristated during a software 'halt' instruction. If, therefore, in a given system a pull-up resistor is used, signal $I/O\bar{M}$ will go high (logic 1) when the microprocessor is halted, erroneously indicating to the system that an I/O instruction is being executed, and (b) No 'WAIT' signal is provided to indicate when the 8085 is in a wait state, as is the case with the 8080.

(v) *Multiplexed Data Bus.* The INTEL 8085 uses a multiplexed data bus. The multiplexing operation is probably best understood by referring to Figures 2.2 and 2.3 on pages 35 and 36. It can be seen from these Figures that during state T_1 of each machine cycle the data bus, *d*, carries no information. Therefore, it follows that in an eight-bit machine with a 16-bit address, eight of the 16 address signals can be output on the data bus during state T_1 and latched before it (the data bus) is used for memory or I/O data. This would release eight pins, that can be used for other purposes. This method of bus-multiplexing is used in the INTEL 8085. In Figure A2.2 we show the time-multiplexing of the data bus during an instruction-fetch cycle, denoted by *M*1 in Figure A2.3. '*ALE*' (*Address Latch Enable*) is a timing pulse generated by the INTEL 8085 in each machine cycle before it enters state *T*2. The trailing edge of *ALE* is set to allow for set up and hold times for the address information. For timing diagrams, if needed, see reference [1]

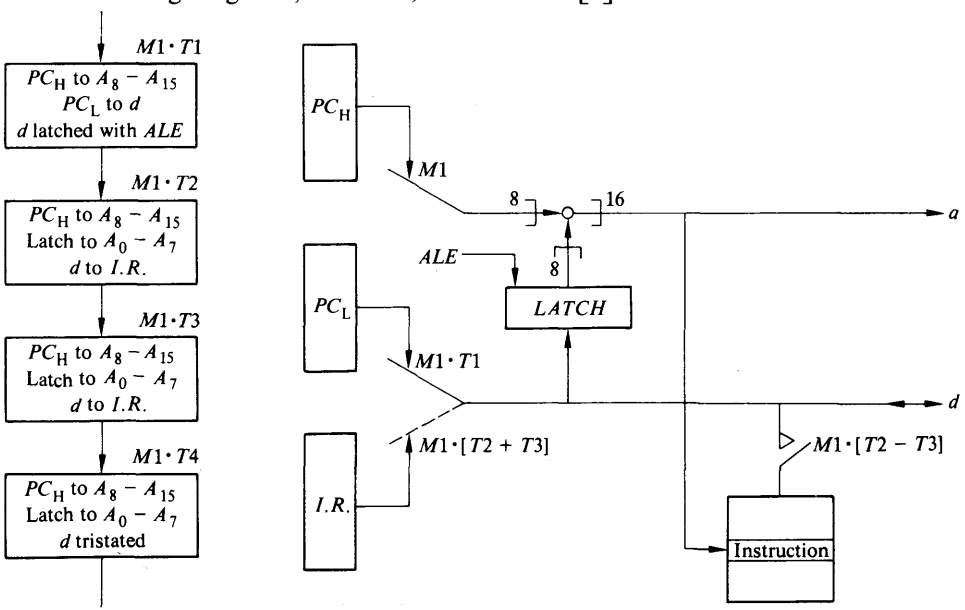


Figure A2.2.

			D6 = 0								D6 = 1								
			$D_2D_1D_0$ = 000	010	110	100	101	111	011	001	001	011	111	101	100	110	010	000	
		000	NOP	Mp = A 002	006	004	005	ROTATE 007	003	001	101	103	107	105	104	rl: = Mhl 106	102	100	B
		010	000	022	026	024	025	027	P: = P + 1 023	P: = H 021	121	123	rl: = r2 127	125	124	rl: = Mhl 126	rl: = r2 122	120	D
		110	SIM	M1: = A 062	066	r: = r + 1 064	r: = r − 1 065	c: = 1 067			161	163	167	165	164	HALT 166	162	160	
		100	RIM	M1: = HL 042	r: = I 046			BCD 047	P: = P − 1 043	P: = HL 041	141	143	147	145	144	146	142	140	H
	0	D5 D4 D3		HL: = M1 052	056	054	055	A: = Ā 057		HL: = HL + P 051	151	153	157	155	154	rl: = Mhl 156	rl: = r2 152	150	L
		101		A: = M1 072	076	074	075	c: = c̄ 077	P: = P − 1 073		171	173	rl: = r2 177	175	174	176	172	170	A
		111		A: = Mp 032	036	034	035	ROTATE 037	P: = P − 1 033	HL: = HL + P 031	131	133	137	135	134	rl: = Mhl 136	132	130	E
		011		012	016	014	015	017	013	011	111	113	117	115	114	116	112	110	C
D_7		001									RET 311		317	CALL 315	314	316	312	310	z
		+ c 001	210	212	216	214	215	217	213	211		INPUT 333	337		334	336	332	330	c
		− c 011	230	232	236	234	235	237	233	231	SP: = HL 371	EI 373	377		374	376	372	370	n
		(-) 111	270	272	276	274	275	277	273	271	PC: = HL 351	353	RESTART 357		354	356	352	350	p
		↓ 101	250	252	256	254	255	257	253	251									
	1	D5 D4 D3				A: = A op r										A: = A op I			
		∧ 100	240	242	246	244	245	247	243	241	341	343	347	345	344	346	342	340	p̄
		∨ 110	260	262	266	264	265	267	263	261	POP p 361	DI 363	367	PUSH P 365	364	366	362	360	n̄
		− 010	220	222	226	224	225	227	223	221	311	OUT 323	327	325	324	326	322	320	c̄
		+ 000	200	202	206	204	205	207	203	201	301	JUMP 303	307	305	304	306	302	300	z̄

	Single Register r	Condition	Operator op	Double Register P	Rotate (A)
000	B	$\bar{z}$ non zero	+	00 BC	Left Logical
010	D	$\bar{c}$ no carry	−	01 DE	Right Logical
110	Mhl	$\bar{n}$ pos or zero	∨	11 SP*	Right Arithmetic
100	H	$\bar{p}$ odd parity	∧	10 HL	Left Arithmatic
101	L	p even parity	↓		
111	A	n negative	compare (z: = A = r; c: = A < r)		
011	E	c carry	− c (A: = A − r − c)		
001	C	z zero	+ c (A: = A − r + c)		*AF in Pop, Push

Program chart for INTEL 8085
(Developed by F. G. Duncan,
University of Bristol, England).

Figure A2.3.

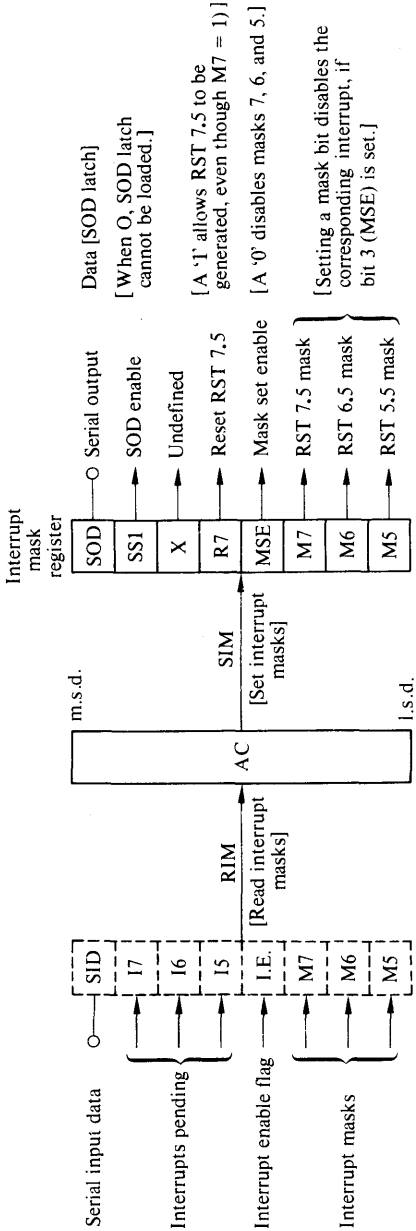

Serial input data
SID
I7
I6
I5
I.E.
M7
M6
M5
Interrupts pending
Interrupt enable flag
Interrupt masks
RIM
[Read interrupt masks]
AC
m.s.d.
l.s.d.
SIM
[Set interrupt masks]
Interrupt mask register
SOD
SS1
X
R7
MSE
M7
M6
M5
Serial output
SOD enable
Undefined
Reset RST 7.5
Mask set enable
RST 7.5 mask
RST 6.5 mask
RST 5.5 mask
Data [SOD latch]
[When O, SOD latch cannot be loaded.]
[A '1' allows RST 7.5 to be generated, even though M7 = 1).]
[A '0' disables masks 7, 6, and 5.]
[Setting a mask bit disables the corresponding interrupt, if bit 3 (MSE) is set.]

Figure A2.4

(vi) *Instruction Set.*† The INTEL 8085 has an identical instruction set as the INTEL 8080, with the exception of two additional instructions, the *SIM* (*Set Interrupt Masks*) and *RIM* (*Read Interrupt Masks*); see Figure A2.3. *SIM* transfers the contents of the *AC* into the Interrupt Mask Register, and *RIM* allows status information to be read into *AC*, as shown in Figure A2.4.

(vii) *Wait/Go Mode.* The design and implementation of 8085 wait/go systems, as with all microprocessors, is straightforward. 8085 wait/go systems are discussed in the next section. The design of wait/go logic would have been somewhat simpler, had a wait signal been made available to us. A 'wait' signal, as we have already explained in Chapter 3, is a signal indicating to the system that the microprocessor has entered a wait (idle) state.

(viii) *Test-and-Skip Mode.* Systems using the test-and-skip mode can be designed and implemented conventionally, as we show in section A2.3.

(ix) *Interrupt Mode.* The interrupt structure provides for (a) A non-maskable direct vectored interrupt, TRAP, on pin 6 (see Figure A2.1). (b) Three direct vectored interrupts, on pins 7, 8 and 9 and (c) Eight indirect vectored interrupts, as in the case of the INTEL 8080.

(x) *D.M.A. Mode.* No special features exist for the design and implementation of the systems using the d.m.a. mode.

(xi) *D.D.T. Mode.* As in the case of the d.m.a. mode, the INTEL 8085 has no special features.

A2.2 WAIT/GO SYSTEMS

In common with all present-day microprocessors, the INTEL 8085 does not execute wait/go cycles. These, as we have already explained in Chapter 3, are I/O cycles with wait/go addresses, denoted by A_w, during which the microprocessor enters the wait state automatically and leaves it when the signal on the 'go' line changes from 0 to 1. It is left to the user to produce a circuit, *the wait/go logic*, which will allow him to initiate wait/go cycles. The block diagram of a wait/go logic is shown in Figure 3.12 on page 63.

Our starting point is the internal state diagram of the INTEL 8085 during the execution of an I/O cycle. This is shown in Figure A2.5. As in the case of the INTEL 8080, we implement a wait/go cycle by causing the microprocessor, when it leaves state $M3{\cdot}T3$, to enter wait state $M3{\cdot}T_w$ instead of state $M3{\cdot}T3$. The transition to state $M3{\cdot}T3$ is to be initiated by a 0 to 1 change on the 'go' line, g, in Figure A2.6.

†The instruction set defines the set of operations that can be performed by a central processor unit.

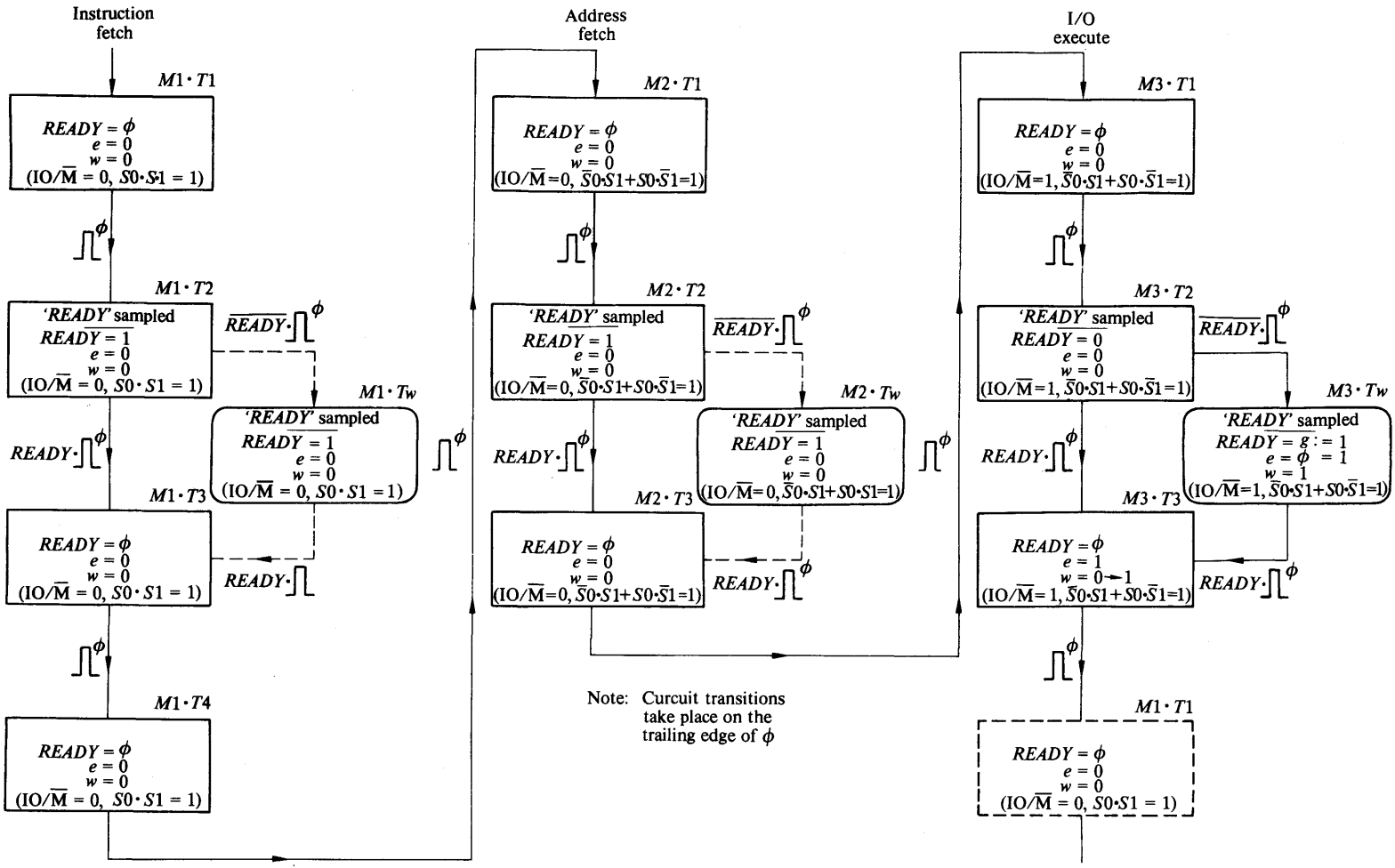

Instruction fetch
M1 · T1
READY = φ
e = 0
w = 0
(IO/M̄ = 0, S0 · S1 = 1)
M1 · T2
'READY' sampled
READY = 1
e = 0
w = 0
(IO/M̄ = 0, S0 · S1 = 1)
READY · φ
M1 · Tw
'READY' sampled
READY = 1
e = 0
w = 0
(IO/M̄ = 0, S0 · S1 = 1)
M1 · T3
READY = φ
e = 0
w = 0
(IO/M̄ = 0, S0 · S1 = 1)
M1 · T4
READY = φ
e = 0
w = 0
(IO/M̄ = 0, S0 · S1 = 1)
Address fetch
M2 · T1
READY = φ
e = 0
w = 0
(IO/M̄ = 0, S̄0 · S1 + S0 · S̄1 = 1)
M2 · T2
'READY' sampled
READY = 1
e = 0
w = 0
(IO/M̄ = 0, S̄0 · S1 + S0 · S̄1 = 1)
M2 · Tw
'READY' sampled
READY = 1
e = 0
w = 0
(IO/M̄ = 0, S̄0 · S1 + S0 · S1 = 1)
M2 · T3
READY = φ
e = 0
w = 0
(IO/M̄ = 0, S̄0 · S1 + S0 · S̄1 = 1)
I/O execute
M3 · T1
READY = φ
e = 0
w = 0
(IO/M̄ = 1, S̄0 · S1 + S0 · S̄1 = 1)
M3 · T2
'READY' sampled
READY = 0
e = 0
w = 0
(IO/M̄ = 1, S̄0 · S1 + S0 · S̄1 = 1)
M3 · Tw
'READY' sampled
READY = g : = 1
e = φ = 1
w = 1
(IO/M̄ = 1, S̄0 · S1 + S0 · S̄1 = 1)
M3 · T3
READY = φ
e = 1
w = 0 → 1
(IO/M̄ = 1, S̄0 · S1 + S0 · S̄1 = 1)
M1 · T1
READY = φ
e = 0
w = 0
(IO/M̄ = 0, S0 · S1 = 1)
Note: Curcuit transitions take place on the trailing edge of φ

Figure A2.5.

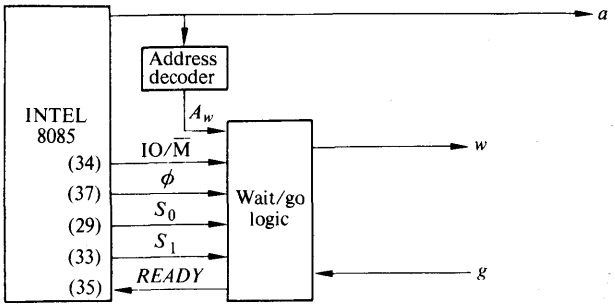


Figure A2.6.

Now, reference to Figure A2.5 shows that the 8085 enters the wait state by pulling its *READY* line low, that is by applying a logic '0' on pin 35. Since this line is sampled during states $T2$ and T_w, we need to generate a logic '0' on the *READY* line when the 8085 enters state $M3 \cdot T2$ and the a bus contains a wait/go address, A_w. The '0' signal on the *READY* line is maintained until the 'go' signal, g, changes from '0' to '1', at which time we must pull the *READY* line high. This allows the microprocessor to move to state $M3 \cdot T3$, at which point it resumes its normal operation. Reference to the m.p.u. chart in Figure A2.1 shows that during the execute cycle of an I/O instruction a '1' is generated on pin 34. This signal is denoted by I/O$\bar{M}$. Also I/O$\bar{M}$ may be pulled high during a software halt, as we explained in section A2.1 (iv). It is, therefore, necessary for our wait/go logic not to look at the I/O$\bar{M}$ terminal during a software 'halt'. Reference to the m.p.u. chart (Figure A2.1) shows that during software 'halt' both the status signals S_0 and S_1, equal to '0'. Therefore, use AND status signal I/O$\bar{M}$ with $\bar{S}_0 \cdot \bar{S}_1$—that is the signal looked at by the wait/go logic is I/O$\bar{M}$ $(S_0 + S_1)$.

Therefore, in addition to signals I/O$\bar{M}$, A_w and ϕ, our wait/go logic must monitor status signals S_0 and S_1. For the sake of clarity we do not show the demultiplexing and multiplexing of the 'w' and 'g' lines. The reader is referred to Figure 3.10 on page 62.

A suitable internal state diagram for our wait/go logic is shown in Figure A2.7. It operates as follows.

The normal state of the circuit is Q0. This state is maintained while the 8085 is active and the peripherals using the wait/go mode are inactive. To maintain the 8085 active we keep its READY line high. In this state our circuit is looking for a wait/go address, A_w. For this purpose it must sample the address bus when the microprocessor is in state $M3 \cdot T1$ in Figure A2.5. Now, when the 8085 is in state $M3 \cdot T1$,

$$\text{I/O}\bar{\text{M}} = 1, \text{ and}$$

$$S_0 + S_1 = 1$$

Thus, we can use signal I/O$\bar{M}(S_0 + S_1)A_w$ to move to the next state.

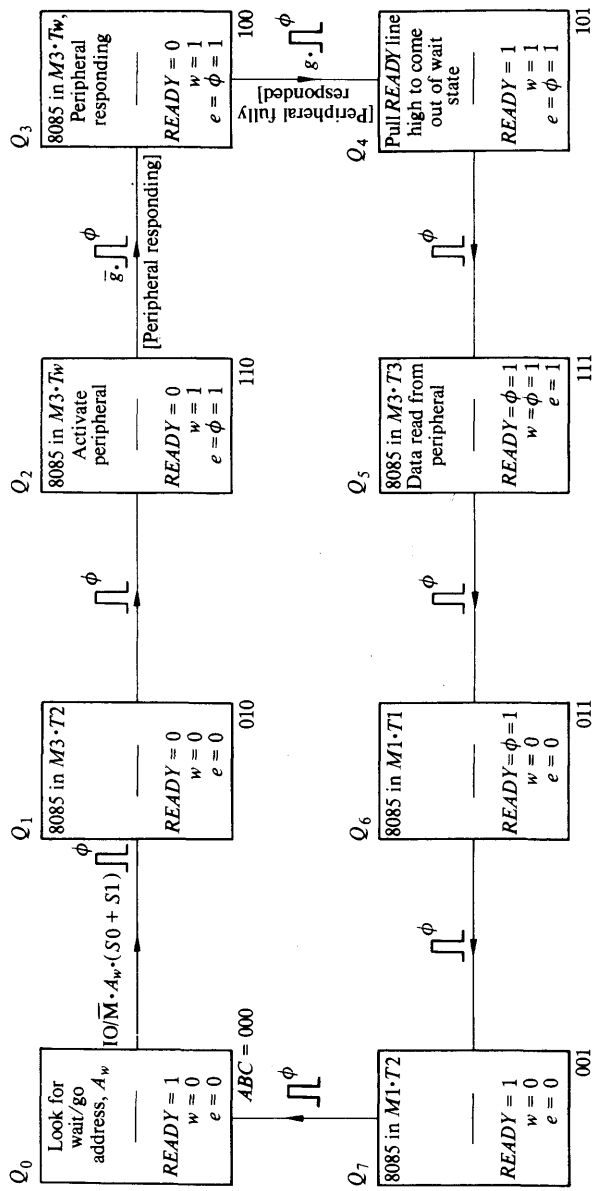

Q_0
Look for wait/go address, A_w
READY = 1
w = 0
e = 0
ABC = 000
001
IO/M · A_w · (S0 + S1)
Q_1
8085 in M3 · T2
READY = 0
w = 0
e = 0
010
Q_2
8085 in M3 · Tw
Activate peripheral
READY = 0
w = 1
e = ϕ = 1
110
[Peripheral responding]
Q_3
8085 in M3 · Tw, Peripheral responding
READY = 0
w = 1
e = ϕ = 1
100
[Peripheral fully responded]
Q_4
Pull READY line high to come out of wait state
READY = 1
w = 1
e = ϕ = 1
101
Q_5
8085 in M3 · T3
Data read from peripheral
READY = ϕ = 1
w = ϕ = 1
e = 1
111
Q_6
8085 in M1 · T1
READY = ϕ = 1
w = 0
e = 0
011
Q_7
8085 in M1 · T2
READY = 1
w = 0
e = 0
001
ϕ
$\bar{g} \cdot$
$g \cdot$

Figure A2.7.

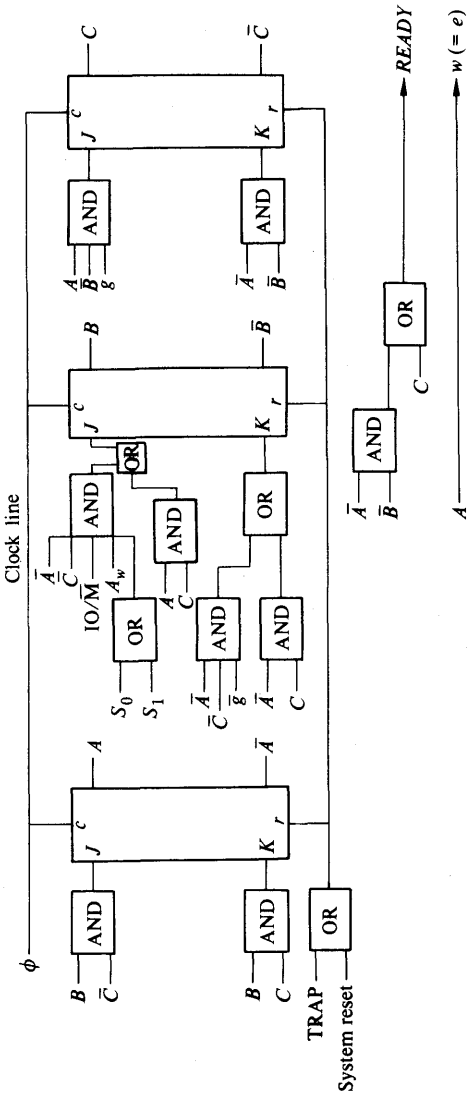

Clock line
φ
AND
OR
J
K
C
r
READY
w (= e)
A
B
C
TRAP
System reset
IO/M
S0
S1

Figure A2.8.

Because circuit transitions in the 8085 chip and in our logic take place on the trailing edge of ϕ, our circuit moves to state $Q1$ at the same time as the microprocessor moves to state $M3{\cdot}T2$ in Figure A2.5. In state $Q1$ we pull the READY signal low. This allows the microprocessor to enter the wait state on the next clock pulse. Our circuit at the same time moves to state $Q2$. At this point we generate our 'wait' signal, w. The change of signal w from 0 to 1 is used to activate the peripheral. To ensure that our circuit does not get 'out of step', we do not move to state $Q3$ until signal g goes low. This tells us that the peripheral began to respond. We remain in state $Q3$ until the peripheral has fully responded, that is until 'go' signal g changes back to 1. In state $Q4$ we pull the READY line high to allow the 8085 to come out of the wait state. It does so on the next clock pulse. At the same time our wait/go logic moves to state $Q5$. Data is read from the accumulator, if the peripheral is a source, when the 8085 is in state $M3{\cdot}T3$. Therefore, in state $Q5$ enable signal e must be high.

From state $Q5$ we can go directly to state Q0. However, this would result in a six-state diagram, that is in a circuit with two unused states. To avoid this situation, for reasons outlined in section 6 of Chapter 1 (page 14), we insert states $Q6$ and $Q7$ in the normal operation of our circuit.

Clearly, the value of the *READY* signal need only be specified in state $Q0$ and when the 8085 is in state $M3{\cdot}T2$ or $M3{\cdot}T_w$. This is because the READY terminal is sampled only in states $T2$ ad T_w (see Figure A2.1).

Because the 8085 enters the wait state during the I/O execute cycle, signal e can be equated to w—see equation 3(a) on page 60.

To avoid unwanted signal spikes on our wait line, w, we use a race-free code to define our eight states. Such a code, derived by direct reference to Figure 1.8, is shown in Figure A2.7. By direct reference to this Figure, we obtain

$$
\begin{aligned}
S_A &= Q1 = \bar{A}B\bar{C}, && \text{therefore } J_A = B\bar{C}\\
R_A &= Q5 = ABC, && \text{therefore } K_A = BC\\
S_B &= Q0{\cdot}\mathrm{I/O}\bar{\mathrm{M}}(S_0+S_1)A_w+Q_4\\
&= \bar{A}\bar{B}\bar{C}{\cdot}\mathrm{I/O}\bar{\mathrm{M}}{\cdot}(S_0+S_1)A_w+A\bar{B}C, && \text{therefore } J_B = \bar{A}\bar{C}{\cdot}\mathrm{I/O}\bar{\mathrm{M}}(S_0+S_1)A_w+AC\\
R_B &= Q2\bar{g}+Q6 = AB\bar{C}\bar{g}+\bar{A}BC, && \text{therefore } K_B = A\bar{C}\bar{g}+\bar{A}C\\
S_c &= Q3g = A\bar{B}\bar{C}g, && \text{therefore } J_C = A\bar{B}g\\
R_c &= Q7 = \bar{A}\bar{B}C, && \text{therefore } K_C = \bar{A}\bar{B}\\
READY &= Q0+Q4+Q7+(Q5)+(Q6)\\
&= \bar{A}\bar{B}\bar{C}+A\bar{B}\bar{C}+\bar{A}\bar{B}C+(ABC)+(\bar{A}BC)\\
&= \bar{A}\bar{B}+C\\
e = w &= Q2+Q3+Q4+(Q5)\\
&= AB\bar{C}+A\bar{B}\bar{C}+A\bar{B}C+(ABC)\\
&= A
\end{aligned}
$$

Reset signal = System reset + $TRAP$

System reset allows our circuit to get in line (to be synchronized) with the rest of the system, and *TRAP* allows it to get out of the wait state in emergencies.

The corresponding circuit is shown in Figure A2.8.

A2.3 TEST-AND-SKIP SYSTEMS

The block diagram of a test-and-skip microprocessor system with one device is shown in Figure 4.2. Its operation is flow-charted in Figure 4.1.

The I/O signals, as in all microprocessors, are generated by executing I/O instructions. In the case of the INTEL 8085 the I/O instructions transfer data in or out of the microprocessor through the Accumulator. Up to 256 input and up to 256 output ports can be directly addressed. Condition flags are not affected by the execution of I/O instructions.

The I/O signals are shown in Figure A2.9. Their timing is shown in Figure A2.10. For the sake of clarity, we are not showing 'rise' and 'fall' times. For detailed timing diagrams, the reader is referred to reference [1].

The block diagram of a test-and-skip system using the INTEL 8085 with one acceptor and one source is shown in A2.11. Signal $rp = 1$ when the source has data available. Similarly, $rq = 1$ when the acceptor can accept data. Tristate signal *ep* is generated by ANDing *Ap*, *Rd* and I/O$\overline{\text{M}}$.

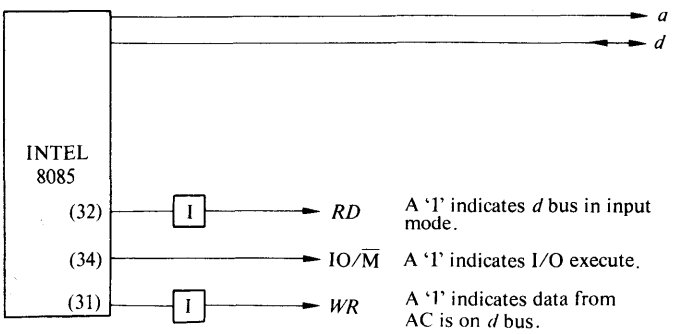


Figure A2.9.

A2.4 INTERRUPT SYSTEMS [1]

The INTEL 8085, as we have already mentioned, has five interrupt inputs. *TRAP*, *RST*7.5, *RST*6.5, *RST*5.5 and *INTR* see Figure A2.12. There are no time constraints on these inputs; they can occur at any time. The interrupt terminals (pins 6, 7, 8, 9, 10 and 11 in Figure A2.12) are sampled during the last clock period

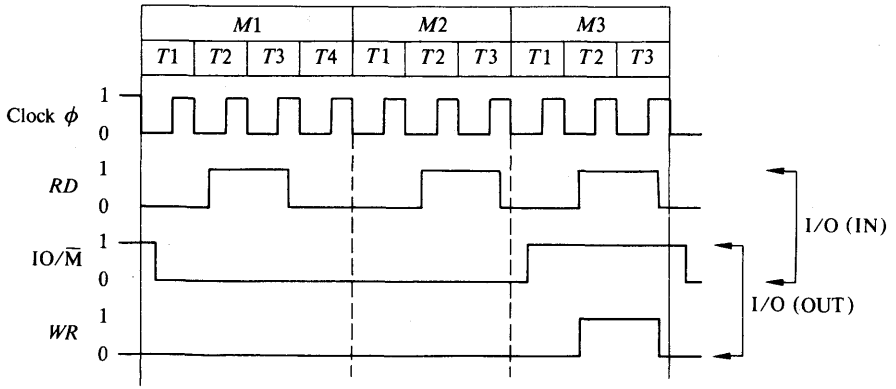


Figure A2.10.

of the instruction that is being executed. They have a fixed priority relative to each other, shown below

TRAP— highest priority
RST 7.5
RST 6.5
RST 5.5
INTR— lowest priority

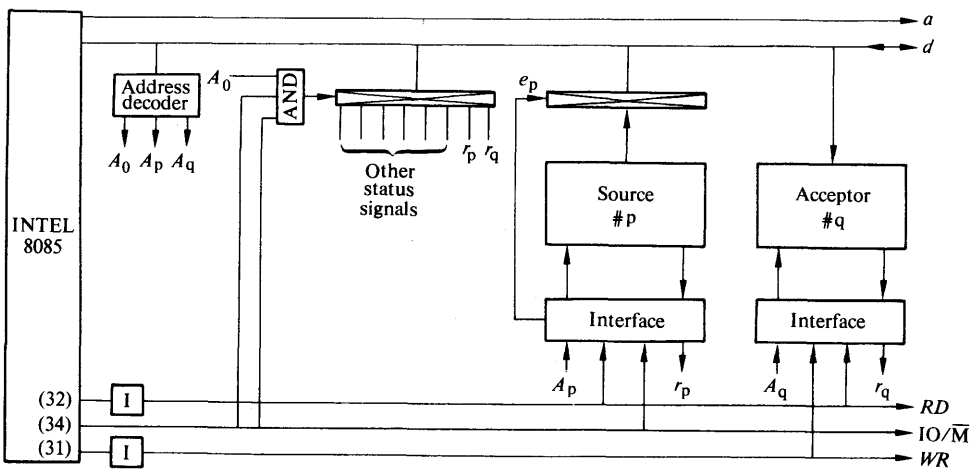


Figure A2.11.

TRAP is a non-maskable *RST* (restart) interrupt, used primarily for emergency situations. This signal must be high when the interrupt terminals are being sampled, but it will not be recognized again until it goes low, then high.

The three direct restarts, RST 7.5, *RST* 6.5 and *RST* 5.5 can be individually masked (disabled) under program control using the *SIM* instruction—see Figure A2.4. Note that *RST* 7.5 request can be set even though its mask is set and the interrupts are disabled. Reference to Figure A2.1 shows that a '0' on pin 36 sets all the *RST* masks, that is it disables pins 7, 8 and 9 in Figure A2.12.

INTR is used as a general purpose interrupt. It is identical to the interrupt input in the 8080; that is, if *INTRA* were the only valid interrupt and if *INTEFF* (interrupt enable flip-flop) is set, the 8085 will reset the flip-flop and enter an interrupt-acknowledge cycle. This is the same as the interrupt cycle of the 8080. This cycle is identical to an instruction fetch cycle with two exceptions. $\overline{INTA}$ is sent out instead of $\overline{RD}$—see Figure A2.10. The address lines are ignored. When $\overline{INTA}$ is sent out, the interrupt logic must provide the op code of an instruction to execute. Although any instruction will do, the logical choice is either a CALL or a RST (restart) instruction. This is because both instructions force the

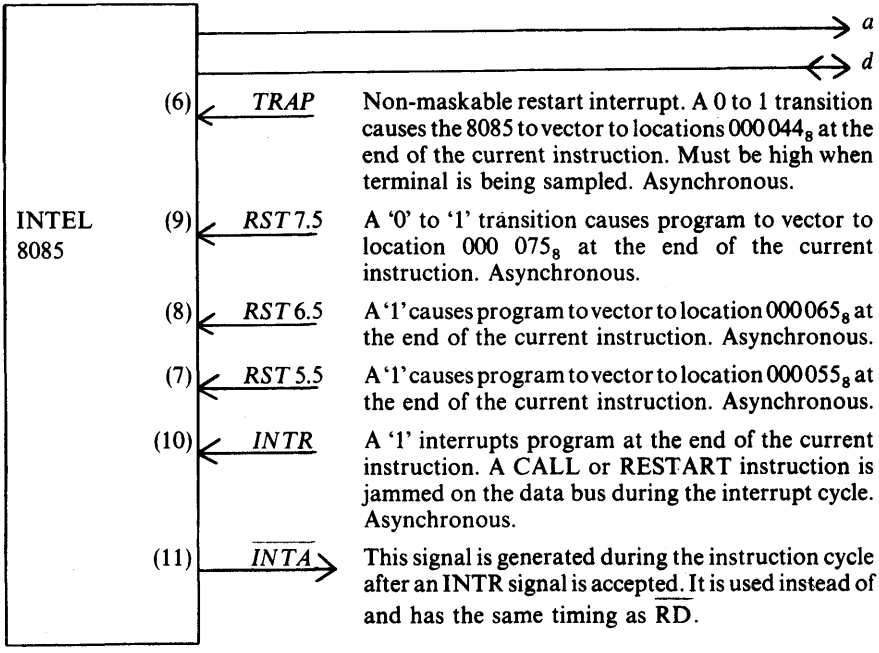


Figure A2.12.

†*INTR* is disabled by *RESET* as well as immediately after an interrupt is accepted.

microprocessor to push the contents of the program counter onto the stack before jumping to a new location. If this statement is not clear, the reader is referred to section 1 in Chapter 5 (pages 109 and 110). Because in the case of the 8080 we have used the *RST* instruction (see section 5.4, page 120), we shall now use the *CALL* instruction.

After receiving the opcode in state $M1 \cdot T3$ in Figure A2.5, the processor decodes it in the next state $M1 \cdot T4$ and determines that two more bytes are required. The 8085 then executes two more machine cycles to obtain the second and third bytes. As in the case of the opcode, bytes 2 and 3 are jammed on the data bus when $\overline{INTA} = 0$. The program counter is not incremented during interrupt acknowledge cycles.

During machine cycles $M4$ and $M5$ the 8085 pushes the upper and then lower bytes of the PC onto the stack and places the two bytes accessed in $M2$ and $M3$ in the lower and upper halves of the program counter. This has the effect of jumping the execution of the program to the location specified by the CALL instruction.

In Figure A2.13 we show a simple arrangement, consisting of a scale-3 pulse counter and three I/O ports, for jamming a CALL instruction during an interrupt cycle. Initially the counter is reset by the $\overline{RD}$ pulses on its reset line. During the interrupt cycle after an INTR (interrupt) signal is accepted, $\overline{RD}$ pulses are suppressed. Instead three $\overline{INTA}$ pulses with the same timing as $\overline{RD}$ are generated. We use these pulses to step up our counter. If our counter is

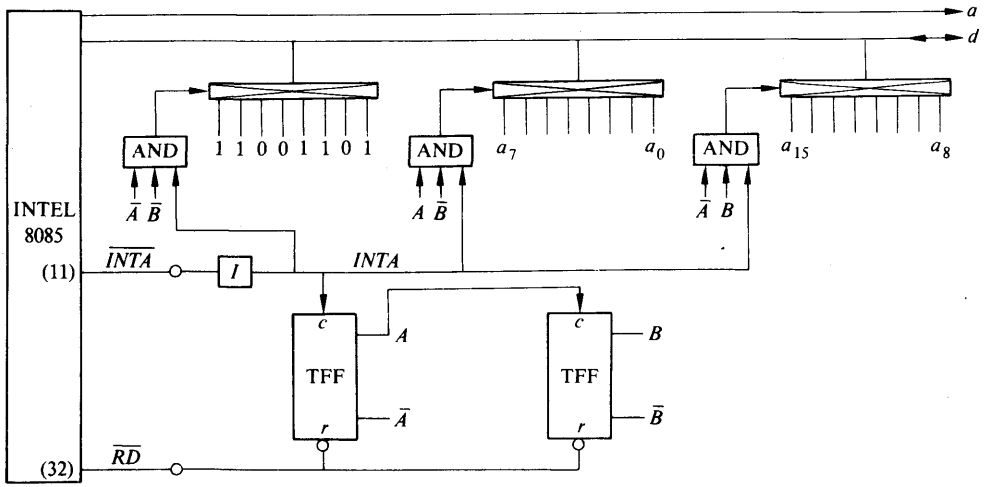


Figure A2.13.

incremented during the trailing edge of a clock pulse, then the correct timing for inserting the three-byte *CALL* instruction is

$\overline{AB}\ \overline{INTA}$—insert opcode 11001101

$A\overline{B}\ \overline{INTA}$—insert byte 2 (low address)

$\overline{AB}\ \overline{INTA}$—insert byte 3 (high address)

The counter resets with the next $\overline{RD}$ pulse.

A2.5 REFERENCE

1. MCS85 User's Manual (Preliminary), Intel Corporation 1976, 1977.

Index

Following his pioneering work in Logic Design, Professor Zissos now presents the field of microprocessors in an accessible and easy-to-read form. Microprocessors have become more and more readily available over the last ten years, bringing computer technology into play in contexts as different as the home, the hospital or the factory. Computers no longer have to be megaliths, expensive, unwieldly and complicated to maintain and operate: the new microprocessor systems, often portable, and using increasingly powerful and compact "chips" at very low cost, are the perfect working alternative to the inscrutable world of traditional computer hardware.

This book is designed to satisfy the current demand for knowledge about how microprocessors work and how they can be used. It is written in the direct, demystified style characteristic of Professor Zissos, and is designed not only for those with a specialist knowledge of electronics but also for the non-specialist (such as system designers, communications experts, non-electronic engineers, physicists, chemists and medical experts who wish to construct their own microprocessor systems.) No previous knowledge of microprocessors is assumed. A special feature of this book is the inclusion of the author's original work with his research assistant J.C. Bathory on the wait/go operation of microprocessors. At the end of each chapter the reader will find a section on 'Problems and Solutions', which illustrate in detail the steps used to design and implement microprocessor systems: all design algorithms are independent of the microprocessors themselves right up to implementation stage, and always work.

This comprehensive account of the subject will be of great interest to those in the fields of telecommunications, medical physics and bio-engineering, digital system design, civil and mechanical engineering, and instrumentation.

Other books by D. Zissos are *Logic Design Algorithms* (1972), *Digital Interface Design* (1974), and *Problems and Solutions in Logic Design* (1976), all published by Oxford University Press.

Academic Press
London New York San Francisco
A Subsidiary of Harcourt Brace Jovanovich, Publishers
24-28 Oval Road, London NW1, England
111 Fifth Avenue, New York, NY 10003, USA
Australian office: PO Box 300, North Ryde, NSW 2113, Australia

Printed in England

0.12.781750.6

Zissos

System Design with Microproces